



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

DEPARTMENT OF COMPUTER SYSTEMS

IMPLEMENTACE 2D ULTRAZVUKOVÝCH SIMULACÍ

IMPLEMENTATION OF 2D ULTRASOUND SIMULATIONS

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

AUTOR PRÁCE

AUTHOR

Bc. DOMINIK ŠIMEK

VEDOUCÍ PRÁCE

SUPERVISOR

doc. Ing. JIŘÍ JAROŠ, Ph.D.

BRNO 2018

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů

Akademický rok 2017/2018

Zadání diplomové práce

Řešitel: **Šimek Dominik, Bc.**

Obor: Počítačové a vestavěné systémy

Téma: **Implementace 2D ultrazvukových simulací**
Implementation of 2D Ultrasound Simulations

Kategorie: Paralelní a distribuované výpočty

Pokyny:

1. Seznamte se se simulačním programem k-Wave a jeho akcelerovanou implementací určenou pro superpočítačové systémy.
2. Osvojte si pokročilé techniky optimalizace kódu pro superpočítače Anselm a Salomon.
3. Analyzujte současnou implementaci 3D ultrazvukové simulace v balíčku k-Wave.
4. Navrhněte a implementujte 2D verzi ultrazvukové simulace jako oddělený program.
5. Navrhněte postup pro sloučení 2D a 3D verze simulace do jednoho simulačního balíčku. Zaměřte se především na automatizované generování kódu pomocí šablon.
6. Navržené řešení implementujte ve zvoleném HPC jazyce.
7. Pomocí jednotkových a regresních testů ověřte správnost implementace.
8. Experimentálně ověřte vlastnosti nové implementace na reálných simulacích.
9. Zhodnoťte dosažené výsledky a diskutujte přínos navržené implementace pro řešení realistických ultrazvukových simulací.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 4 zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese

<http://www.fit.vutbr.cz/info/szz/>

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).


Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Jaroš Jiří, doc. Ing., Ph.D., UPSY FIT VUT**

Datum zadání: 1. listopadu 2017

Datum odevzdání: 23. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
602 00 Brno, Božetěchova 2



prof. Ing. Lukáš Sekanina, Ph.D.
vedoucí ústavu

Abstrakt

Práca sa zaoberá návrhom a implementáciou 2D simulácie ultrazvukových vĺn. Simulácia ultrazvuku nachádza svoje uplatnenie v medicíne, biofyzike či rekonštrukcii obrazu. Ako príklad môžeme uviesť použitie fokusovaného ultrazvuku na diagnostiku a liečbu rakoviny. Program je súčasťou simulačného balíka k-Wave určeného pre superpočítačové systémy, konkrétne stroje s architektúrou zdieľaného adresového priestoru. Program je implementovaný v jazyku C++ s využitím akcelerácie pomocou OpenMP. Pomocou implementovaného riešenia je možné riešiť simulácie veľkých rozmerov v 2D priestore. Práca sa ďalej zaoberá zjednotením kódu 2D a 3D simulácie pomocou moderných prostriedkov C++. Reálnym príkladom využitia je simulácia ultrazvuku pri transkraniálnej neuromodulácii a neurostimulácii, ktorá prebieha v doménach o veľkosti 16384^2 (a viac) bodov mriežky. Simulácia takýchto rozmerov môže pri použití pôvodnej MATLAB 2D k-Wave trvať niekoľko dní. Implementované riešenie dosahuje voči MATLAB 2D k-Wave 7 až 8 násobné zrýchlenie na superpočítačoch Anselm a Salomon.

Abstract

The work deals with design and implementation of 2D ultrasound simulation. Applications of the ultrasound simulation can be found in medicine, biophysic or image reconstruction. As an example of using the ultrasound simulation we can mention High Intensity Focused Ultrasound that is used for diagnosing and treating cancer. The program is part of the k-Wave toolbox designed for supercomputer systems, specifically for machines with shared memory architecture. The program is implemented in the C++ language and using OpenMP acceleration. Using the designed solution, it is possible to solve large-scale simulations in 2D space. The work also deals with merging and unification of the 2D and 3D simulation using modern C++. A realistic example of use is ultrasound simulation in transcranial neuromodulation and neurostimulation in large domains, which have more than 16384^2 grid points. Simulation of such size may take several days if we use the original MATLAB 2D k-Wave. Speedup of the new implementation is up to 8 on the Anselm and Salomon supercomputers.

Klíčové slová

k-Wave, 2D ultrazvuková simulácia, 3D ultrazvuková simulácia, C++, OpenMP, superpočítač, generické programovanie.

Keywords

k-Wave, 2D ultrasound simulation, 3D ultrasound simulation, C++, OpenMP, supercomputer, generic programming.

Citácia

ŠIMEK, Dominik. *Implementace 2D ultrazvukových simulací*. Brno, 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Ing. Jiří Jaroš, Ph.D.

Implementace 2D ultrazvukových simulací

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením pána doc. Ing. Jiřího Jaroša, Ph.D. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Dominik Šimek
21. mája 2018

Podakovanie

Touto cestou by som chcel veľmi pekne poďakovať vedúcemu mojej práce, pánovi doc. Ing. Jiřímu Jarošovi, Ph.D. za jeho čas, veľkú ochotu a najmä cenné rady, ktoré mi poskytoval počas celej doby spolupráce. Chcel by som mu poďakovať za jeho priateľský, trpezlivý a hlavne odborný prístup, bez ktorého by sa táto práca tvorila veľmi ťažko.

Veľká vďaka patrí mojej drahej rodine, mojej priateľke Frederike Michalkovej a všetkým blízkym, ktorý ma počas celého štúdia podporovali a pomáhali mi.

Táto práca bola podporená Ministerstvom školstva, mládeže a telovýchovy z podpory Veľkých infraštruktúr pre výskum, experimentálny vývoj a inováciu v rámci projektu IT4Innovations národné superpočítačové centrum - LM2015070.

Obsah

1	Úvod	3
2	Projekt k-Wave	5
2.1	Princíp fungovania	5
2.2	C++ implementácia	6
2.2.1	Tok programu (WorkFlow)	7
2.2.2	Štruktúra kódu	7
2.2.3	Optimalizácia kódu	12
2.2.4	Implementácia k-Wave využívajúca zdieľanú pamäť	13
2.3	Príklad spustenia programu	14
2.4	Prípady použitia	15
3	Superpočítače	16
3.1	Architektúra superpočítača	16
3.1.1	Architektúra jedného uzla	16
3.1.2	Prepojenie uzlov	17
3.2	Plánovanie úloh	17
3.3	Superpočítač Salomon	18
3.3.1	Modulárny systém	19
3.3.2	Výpočtové uzly	19
4	Paralelný výpočet	21
4.1	Zdieľaný adresový priestor	22
4.2	Zasielanie správ	22
4.3	Dátovo paralelný model	23
4.4	Škálovanie	24
5	Prostriedky pre implementáciu	25
5.1	Prostriedky C++	25
5.1.1	Dedičnosť	25
5.1.2	Generické programovanie	26
5.2	OpenMP	27
5.2.1	Implementácia OpenMP	28
5.2.2	Princípy fungovania OpenMP	28
5.2.3	Komponenty OpenMP	29
5.3	Knižnica HDF5	31
5.4	Knižnica FFTW	32
5.5	Knižnica Google Test	32

5.6	Kompilátory	33
5.7	Generovanie kódu za behu programu	33
5.8	Dokumentácia	33
6	Implementácia 2D k-Wave	34
6.1	Návrh riešenia	34
6.1.1	Trieda MatrixContainer	34
6.1.2	Trieda OutputStreamContainer	34
6.1.3	Trieda FftwComplexMatrix	35
6.1.4	Trieda KSpaceFirstOrder3DSolver	35
6.2	Overenie funkčnosti a meranie výkonnosti	36
7	Zjednotená implementácia 2D/3D k-Wave	37
7.1	Návrh riešenia	37
7.1.1	Trieda MatrixContainer	37
7.1.2	Trieda OutputStreamContainer	38
7.1.3	Trieda FftwComplexMatrix	38
7.1.4	Trieda KSpaceFirstOrderSolver	38
7.1.5	Vylepšenia triedy KSpaceFirstOrderSolver	41
7.2	Overenie funkčnosti a meranie výkonnosti	45
8	Experimentálne výsledky	46
8.1	Doba výpočtu jedného kroku simulácie	46
8.1.1	Vplyv rozmerov domény na čas výpočtu	48
8.2	Škálovanie výpočtu	48
8.3	Spotreba pamäte	52
8.4	Analýza kritických častí výpočtu	53
8.5	Vplyv frekvencie procesoru na rýchlosť výpočtu	55
8.6	Vizualizácia výsledkov simulácie	55
9	Testovanie	58
9.1	Jednotkové testy	58
9.2	Integračné testy	59
10	Záver	61
	Literatúra	63
A	Prepínače programu kspaceFirstOrder3D-OMP	65
B	Príklad výstupu programu kspaceFirstOrder3D-OMP	67
C	Jednotkové testy	69
D	Formát vstupného a výstupného HDF5 súboru	73
E	Popis obsahu CD	79

Kapitola 1

Úvod

V dnešnej dobe väčšina ľudskej odbornej činnosti vyžaduje veľmi presné, rýchle a ekonomické riešenia. Výnimkou nie je ani medicína, ktorú je nutné vzhľadom na zdravotný stav populácie modernizovať a zdostupňovať. Podľa Svetovej Zdravotníckej Organizácie [20] bolo v roku 2012 celosvetovo diagnostikovaných viac ako 14 miliónov nových prípadov rakoviny, pričom počet nových pacientov postupne rastie. Faktom je, že rakovina je bohužiaľ príčinou takmer 1 z 6 úmrtí ročne [20]. Existuje viacero metód liečenia nádorových ochorení, napríklad chirurgická operácia, chemoterapia a rádioterapia (rádioaktívne žiarenie). Uvedené metódy ale majú niekoľko nedostatkov a nežiadúcich účinkov (rádioaktívne žiarenie ničí aj zdravé bunky, agresivita chemoterapie, dlhé zotavovanie), ktoré znižujú úspešnosť liečby [5].

Dôležitou súčasťou liečby je okrem odstránenia tumoru aj včasná diagnostika. V tomto smere sa moderná medicína začína prelínať s biofyzikou, ktorá poskytuje sľubnú alternatívu k tradičným metódam. Metóda použitia ultrazvuku s vysokou intenzitou (high intensity focused ultrasound – HIFU) [1] je neinvazívny prístup použiteľný na liečbu rakoviny. Technika HIFU je založená na vysielaní cieleného ultrazvukového lúča do tkaniva pomocou ultrazvukového vysielača. V centre žiarenia je energia ultrazvuku dostatočná nato, aby usmrtila nádorové bunky, zatiaľčo okolie zachováva nepoškodené. Problémom tejto metódy je jej výpočtová a pamäťová náročnosť. Napríklad simulácia v doméne $25 \times 25 \times 25$ cm môže vyžadovať aj viac ako 10^{12} bodov mriežky. Simulácie takýchto rozmerov sú prakticky neriešiteľné. Je teda nutné nájsť prístup, ktorý umožní presný, škálovateľný a ekonomický výpočet. Jednou z možností je použitie projektu k-Wave [16], ktorý rieši simuláciu ultrazvuku v 1D, 2D a 3D priestore.

Ďalším využitím ultrazvuku v medicínskom odvetví je transkraniálna ultrazvuková neuromodulácia a stimulácia (Ultrasonic modulation and stimulation – UNMS) [12]. Na rozdiel od tradičných metód stimulácie mozgu pomocou implantovaných elektród je stimulácia ultrazvukom menej riskantná, neinvazívna metóda. Alternatívnou neinvazívnou metódou je napríklad transkraniálna magnetická neurostimulácia, ktorá je prakticky použiteľná, ale má isté obmedzenia. Problémom je zaostrenie na veľmi malú oblasť mozgu a prienik žiarenia do okolitého tkaniva. UNMS v poslednej dobe vzbudila veľký záujem vedcov a výskumníkov, pretože ponúka potencionálne riešenia nedostatkov iných metód. Samozrejme UNMS má aj negatíva, najmä výpočtovú a pamäťovú náročnosť. 3D Simulácia s veľkosťou 1024^3 bodov mriežky trvala na Superpočítači Salomon viac ako 110 hodín [12]. Veľkosti riešenej domény sa však môžu vyšplhať aj na 16384^3 , čo robí simuláciu časovo a finančne nereálnou. Našťastie je možné niektoré problémy riešiť aj v 2D priestore, čo poskytuje možnosť simulácií

veľkých rozmerov (16384^2), ktorú je možné riešiť v prijateľnom čase. Simulácie ktoré by boli v 3D prakticky neriešiteľné sú v 2D vypočítané za podstatne kratší čas.

Projekt k-Wave [17] je používaný v oboch vyššie uvedených prípadoch. Program bol pôvodne vytvorený pre prostredie *MATLAB*¹. 3D kód bol kvôli extrémnej výpočtovej náročnosti postupne implementovaný v jazyku *C++*² za použitia rôznych HPC (angl. high performance computing) jazykov. Keďže sa objavili nové možnosti využitia 2D simulácií, nastala potreba urýchlenia 2D k-Wave. Cieľom tejto diplomovej práce je teda implementácia 2D verzie k-Wave v *C++* s použitím vhodného HPC jazyka. Požiadavky na 2D implementáciu sú najmä presnosť, efektivita, možnosť využitia prostriedkov superpočítača a tým pádom riešenie extrémne veľkých úloh.

Ďalším cieľom tejto diplomovej práce je zjednotenie 2D a 3D *C++* kódu do jedného, prehľadného a výkonného celku. Tento krok by mal v budúcnosti pomôcť vývojárom projektu k-Wave v integrácii viacerých verzií programu a v možnosti jednoduchšej rozširiteľnosti. V novej, zjednotenej verzii kódu budú využité moderné prostriedky *C++*, šablóny, generické programovanie a iné. Novovytvorený program bude podrobený dôsledným testom presnosti výpočtu, jeho efektivity a celkového výkonu. Práca sa bude primárne zaoberať simuláciou ultrazvuku v mäkkých tkanivách.

¹<https://www.mathworks.com/products/matlab.html>

²<http://en.cppreference.com/w/>

Kapitola 2

Projekt k-Wave

Projekt k-Wave¹ je sada nástrojov (pôvodne *MATLAB*) vyvinutých na simuláciu šírenia ultrazvukových a akustických vln v 1D, 2D a 3D [16]. Nástroj k-Wave je medzinárodným projektom vyvíjaným Dr. Bradley E. Treeby a Dr. Ben Cox z University College London a doc. Dr. Jiří Jaroš z Vysoké učení technické v Brně. Projekt k-Wave je vyvíjaný a distribuovaný ako otvorený software².

2.1 Princíp fungovania

Nástroj k-Wave má veľký rozsah funkcionality, princíp simulácie je ale založený na pokročilom numerickom modeli [17], [18], [16]. Akustický model môže definovať lineárne a nelineárne šírenie vlny, homogénny a heterogénny popis materiálu a akustickú absorpciu energie. Model umožňuje definovať rôzne zdroje tlaku a rýchlosti akustických vln (fotoakustické zdroje, ultrazvukové vysielače). Ďalej je možné modelovať plochy, na ktorých sa budú požadované veličiny vzorkovať (rýchlosť, tlak, intenzita). Okrem samotných výpočtov je možná aj vizualizácia výsledkov.

Numerický model je založený na riešení troch parciálne diferenciálnych rovníc prvého rádu. Rovnice sú riešené použitím k -priestorových pseudospektrálnych metód. Tri fundamentálne rovnice riešené v numerickom modeli sú výpočet zmeny rýchlosti 2.1, výpočet zmeny hustoty 2.2 (zákon zachovania hmotnosti) a výpočet zmeny tlaku 2.3. V uvedených rovniciach je u akustická rýchlosť častice, p akustický tlak, ρ akustická hustota, ρ_0 akustická hustota okolia, c_0 rýchlosť šírenia zvuku (rovnaká vo všetkých smeroch), d vzdialenosť častíc a L Laplaceov operátor.

$$\frac{\partial u}{\partial t} = -\frac{1}{\rho_0} \nabla p \quad (2.1)$$

Výpočet zmeny rýchlosti.

Priestorové gradienty sú počítané pomocou Fourierovej kolokačnej metódy, zatiaľčo časové gradienty pomocou metódy konečných diferencií. Absorpcia energie je vypočítaná pomocou lineárneho, integro – diferenciálneho operátora založeného na frakčnom Laplaceovom operátore. Pre absorpciu vln na okrajoch domény je použité "PML" (angl. perfectly mat-

¹<http://www.k-wave.org>

²<http://www.k-wave.org/license.php>

$$\frac{\partial \rho}{\partial t} = -(2\rho + \rho_0)\nabla \cdot u \quad (2.2)$$

Výpočet zmeny hmotnosti.

$$p = c_0^2(\rho + \frac{B}{2A} \frac{\rho^2}{\rho_0} - L_\rho) \quad (2.3)$$

Výpočet zmeny tlaku.

ched layer). Oproti modelu "FDTD" (angl. finite-difference time domain) má použitý numerický model výhodu najmä v tom, že potrebuje menej časových a priestorových bodov, pri zachovaní rovnako presných výsledkov. To má za následok, že simulácia beží rýchlejšie a spotrebovávajú menej pamäte.

Dôležitým faktom je, že rôzne vlastnosti média ovplyvňujú náročnosť simulácie (časovú aj pamäťovú). Jednou z najzásadnejších vlastností média je jeho homogenita, resp. heterogenita. Napríklad pri výpočte tlaku sa u homogénnej varianty počíta len so skalárnymi hodnotami (hustota média, rýchlosť šírenia zvuku), u heterogénnej varianty s maticami hodnôt (nezanedbateľne sa zvyšuje náročnosť výpočtu). Stratové/bezstratové médium (absorpcia) sa počíta nad rovnakou dátovou štruktúrou, ale u bezstratového média sa vynechajú niektoré výpočty, vykonáva sa menej operácií, výpočet je tým pádom rýchlejší ako u média stratového. Linearita alebo nelinearita akustickej vlny nemá zásadný vplyv na rýchlosť výpočtu či spotrebovanú pamäť.

Dimenzionalita domény má tak isto vplyv na vlastnosti simulácie. Logicky, 1D, 2D a 3D simulácie sa počítajú nad inými dátovými štruktúrami, pričom sa vykonáva rôzny počet operácií, s rôznymi pamäťovými nárokmi. Okrem toho to zavádza rôzne procesné zmeny, čo má za následok, že každá verzia k-Wave (1D, 2D, 3D) má vlastný zdrojový kód. Aj z tohto dôvodu bolo jedným z cieľov tejto práce zjednotenie 2D a 3D kódu (C++).

2.2 C++ implementácia

Program k-Wave bol pôvodne implementovaný v prostredí *MATLAB*, do ktorého bol integrovaný ako sada nástrojov (angl. toolbox). Táto implementácia zahŕňala simulácie v 1D, 2D a 3D priestore. Efektivita *MATLAB* kódu nebola dostačujúca na vykonávanie simulácie vo veľkých doménach (viac ako 1024^3 bodov), resp. jednalo sa o neekonomické a časovo náročné riešenie. Z tohto dôvodu bola neskôr 3D simulácia implementovaná v jazyku C++. Nová implementácia využíva výkonnosť natívneho kódu, pričom je zavedená podpora rôznych HPC jazykov (resp. rozšírení jazykov, prípadne iných HPC prostriedkov určených pre optimalizáciu a akceleráciu kódu). Vznikli teda rôzne C++ implementácie (optimalizované pre rôzne platformy a veľkosti domén) s využitím prostriedkov paralelného spracovania na úrovni vlákien/procesov, konkrétne *OpenMP*³, *MPI*⁴ a *Nvidia CUDA*⁵. Okrem toho sú jed-

³<http://www.openmp.org/>

⁴<http://mpi-forum.org/docs/>

⁵<https://developer.nvidia.com/cuda-zone>

notlivé výpočtové jadrá optimalizované pre vektorové spracovanie dát pomocou *OpenMP*, *Intel SSE*⁶, *Intel AVX*⁷, *SIMT*⁸.

2.2.1 Tok programu (Workflow)

Program môžeme rozdeliť na niekoľko základných krokov (fáze programu), ktoré sa vykonávajú sekvenčne, za sebou. Tok týchto častí programu je zobrazený na Obrázku 2.1. Pred spustením simulácie je nutné vygenerovať dáta, ktoré budú tvoriť vstup simulácie (napr. sken pacienta metódou *CT*⁹). Ďalej je nutné nastaviť vlastnosti simulácie, ktorými sú parametre média, parametre akustickej vlny, počet krokov simulácie, počet vlákien programu, atď.

Po spustení programu sa najskôr načítajú parametre simulácie (1.). Ak bol program spustený správne, pokračuje sa alokáciou pamäte (2.). V tomto kroku sa zistí, či je simulácia z hľadiska pamäťových nárokov na danom stroji možná. Po úspešnej alokácii pamäte sa pokračuje načítaním vstupného súboru (3.) do pamäte alokovanej v predchádzajúcom kroku.

Ďalším krokom je inicializácia knižnice použitej pre výpočet FFT, resp. vytvorenie vytvorenie plánov (4.) pre výpočet 1D, 2D a 3D FFT. V ďalšom kroku sa vykonáva predspracovanie vstupných dát (5.), čo zahŕňa najmä prepočítanie indexov matíc z notácie *MATLAB* do notácie *C++* a generovanie potrebných konštánt. Krokom číslo 6. je hlavný simulačný cyklus (výpočtovo najnáročnejšia časť programu). V tejto časti sa postupne volajú jednotlivé výpočtové jadrá podľa zadaných parametrov simulácie. Celý výpočet sa spúšťa toľko krát, koľko je požadovaný počet krokov simulácie.

Po vykonaní všetkých krokov simulácie prichádza na rad spracovanie dát (7.). V tejto časti sú uložené masky senzorov, spracované a uzatvorené výstupné prúdy. V prípade potreby sú *C++* indexy polí prepočítané na indexy polí *MATLAB*. Posledné kroky sú uloženie výstupných dát simulácie (8.) a uvoľnenie pamäte (9.).

Simulácia sa vykonáva po krokoch. Každý krok simulácie zahŕňa množstvo výpočtov, z vysokej úrovne ho popisuje Obrázok 2.2. Prvou časťou je výpočet *FFT* nad maticou dát (transformácia reálnych dát na komplexné). Tento výpočet je realizovaný pomocou rutín knižnice *FFTW* 5.4. Ďalej nasledujú operácie s jednotlivými elementami matice. Tieto operácie sú vykonávané vo frekvenčnej doméne. Poslednou časťou je výpočet inverznej Fourierovej transformácie (*IFFT*), kde sa komplexné dáta transformujú na reálne.

2.2.2 Štruktúra kódu

C++ kód k-Wave je štruktúrovaný do modulov, kde je každý modul definovaný ako *C++* trieda. Každá trieda má deklaráciu vo vlastnom hlavičkovom súbore (.h) a implementáciu v samostatnom zdrojovom súbore (.cpp).

Trieda *KSpaceFirstOrder3DSolver*

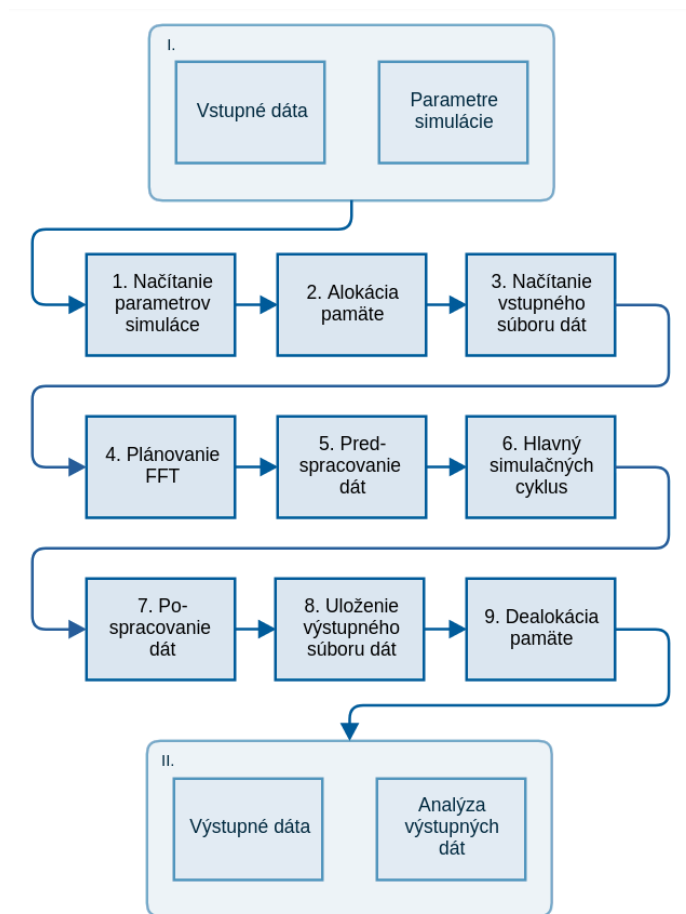
Hlavná trieda simulačného programu, riadi celý proces simulácie. Ako členské premenné obsahuje objekty typu *MatrixContainer*, *OutputStreamContainer*, parametre simulácie

⁶<https://www.intel.com/content/www/us/en/support/articles/000005779/processors.html>

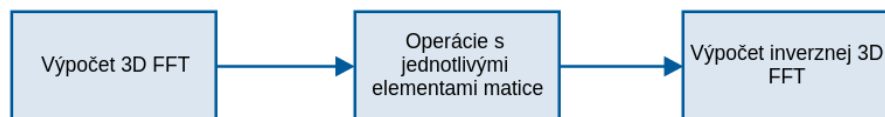
⁷<https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions>

⁸SIMT – technológia využívaná v architektúrach GPU, skupina vlákien vykonáva 1 inštrukciu (angl. Single Instruction Multiple Threads)

⁹Počítačová tomografia (angl. computer tomography)

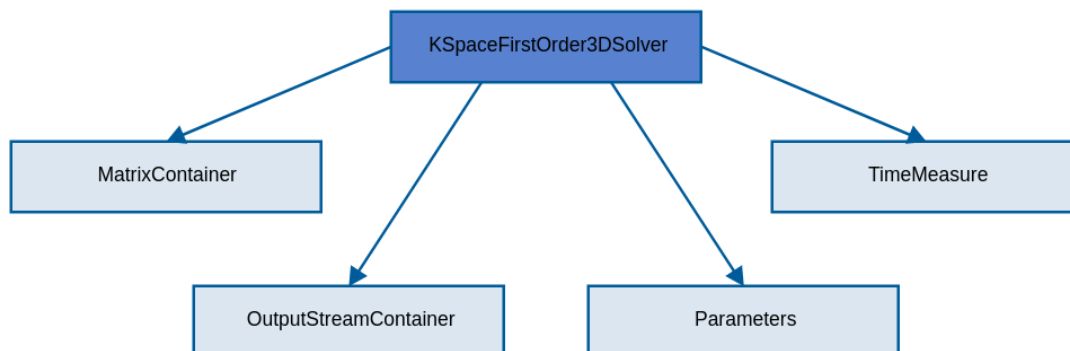


Obr. 2.1: Diagram toku práce programu.



Obr. 2.2: Vysoko úrovňový pohľad na simulačný krok, výpočet gradientu matice.

(`Parameters`), časy jednotlivých častí výpočtu (`TimeMeasure`) a ďalšie premenné uchovávajúce stav simulácie. Všetky členské premenné triedy sú privátne. Na Obrázku 2.3 je možné vidieť diagram tried, ktoré sú členmi triedy `KSpaceFirstOrder3DSolver` (zjednodušený diagram spolupráce tried, zobrazuje najvýznamnejšie triedy).



Obr. 2.3: Diagram spolupráce pre triedu `KSpaceFirstOrder3DSolver`.

Verejné metódy sú metódy určené na inicializáciu simulácie, jej spustenie a získanie štatistík. Samotná simulácia sa spúšťa metódou `compute`. Všetky ostatné metódy (metóda predspracovania dát, samotné výpočtové jadrá, získavanie jednotlivých matic z kontajnera, uloženie dát senzorov, metóda finálneho spracovania dát) sú implementované ako chránené ("protected") a používané len vrámci triedy `KSpaceFirstOrder3DSolver`. Metóda, ktorá implementuje hlavný cyklus programu a riadi celú simuláciu je `computeMainLoop`.

Kritické funkcie jazyka MATLAB operujúce nad veľkým množstvom dát sú v `C++` riešené buď volaním funkcií z externých knižníc (optimalizovaných), alebo sú implementované samostatne v takzvaných výpočtových jadrách (angl. compute kernels). Výpočtové jadrá sú spracovávané paralelne a optimalizované pre vysoký výkon. Spôsob optimalizácie jednotlivých funkcií závisí na použitom HPC jazyku. Táto problematika bude bližšie rozobraná v Kapitole 2.2.3, ktorá sa venuje optimalizácii kódu v projekte k-Wave.

V triede je implementované meranie času jednotlivých fáz výpočtu (resp. celej simulácie), ktoré sú priebežne zobrazované užívateľovi. Užívateľ má teda počas celej doby výpočtu prehľad o tom, v akom stave sa simulácia nachádza a koľko času zostáva na dokončenie výpočtu. Na konci simulácie sú zobrazené finálne štatistiky zahŕňajúce časy jednotlivých častí programu (alokácia pamäte, načítanie dát, FFT plánovanie, výpočet, uloženie výstupných dát, ...), spotrebu pamäte apod.

Trieda `MatrixContainer`

Trieda uchováajúca všetky matice potrebné pre simuláciu, zaberá teda prevažnú väčšinu spotrebovanej pamäte celého programu. Kontajner pre matice je definovaný ako objekt `C++` šablóny `std::map<MatrixIdx, MatrixRecord>`, kde `MatrixIdx` je index matice (identifikuje maticu v kontajneri) a `MatrixRecord` je záznam uchováajúci samotnú maticu (v kontajnery sú uložené matice pre rôzne účely).

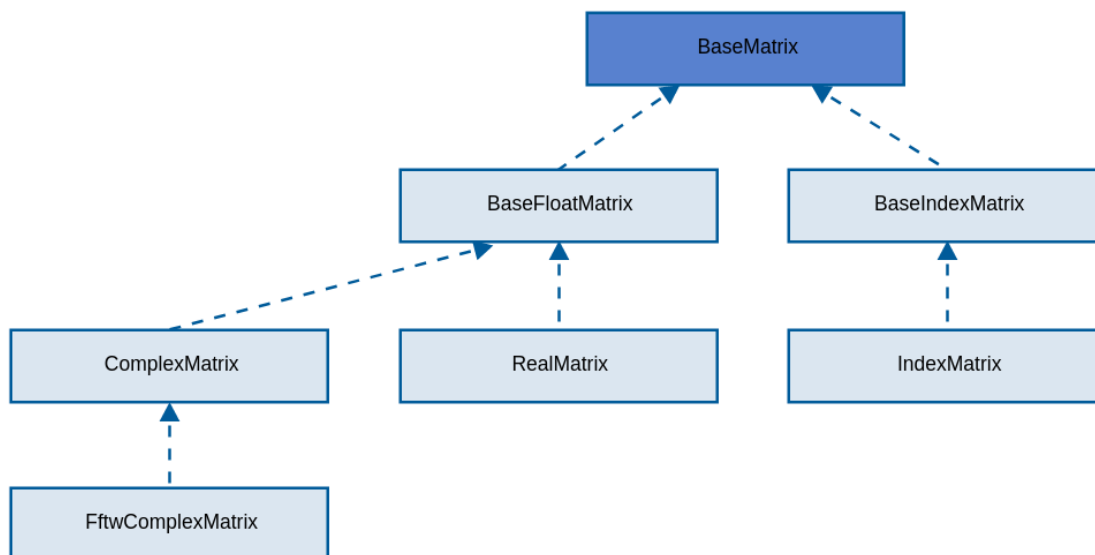
Trieda implementuje verejné metódy pre pridanie matice do kontajnera, vytvorenie matice (alokácia pamäte, resp. volanie konštruktora matice daného typu), uvoľnenie pamäte, načítanie vstupných dát zo súboru apod. Okrem toho implementuje metódu `getMatrix(const`

`MatrixIdx matrixIdx`), ktorá na základe identifikátora vráti ľubovoľnú maticu z kontajnera.

Maticové triedy

V projekte sú implementované triedy matíc rôznych druhov v závislosti na dátovom type, ktorý bude v matici uložený. Základnou triedou, z ktorej dedia všetky ostatné maticové triedy je trieda `BaseMatrix`. Táto trieda definuje základné rozhranie pre prácu s maticami. Trieda `BaseFloatMatrix` je odvodená od `BaseMatrix` a tvorí základ pre triedy `RealMatrix`, `ComplexMatrix` a `FftwComplexMatrix`. Prvé dve z uvedených matíc uchovávajú reálne, resp. komplexné čísla typu `float`. Posledná z uvedených matíc uchováva komplexné čísla a navyše implementuje metódy, ktoré vykonávajú nad prvkami matice 1D, 2D a 3D *FFT*.

Trieda `BaseIndexMatrix` tvorí základ pre maticu `IndexMatrix`, v ktorej nie sú uložené priamo hodnoty (napr. reálne dáta), ale indexy prvkov uložených v iných maticiach (napr. reálnych). Tieto indexy definujú súradnice senzorov (vzorkovanie) a zdrojov (vysielače). Obrázok 2.4 zobrazuje hierarchiu dedičnosti maticových tried.



Obr. 2.4: Diagram dedičnosti maticových tried.

Trieda `OutputStreamContainer`

Trieda uchováajúca všetky výstupné prúdy (angl. output streams) určené na ukladanie vzorkovaných dát. Kontajner pre výstupné prúdy je definovaný ako objekt `std::map<OutputStreamIdx, BaseOutputStream*>`, kde `OutputStreamIdx` je index prúdu (identifikuje výstupný prúd v kontajneri) a `BaseOutputStream` je základná trieda definujúca rozhranie objektov typu `OutputStream`.

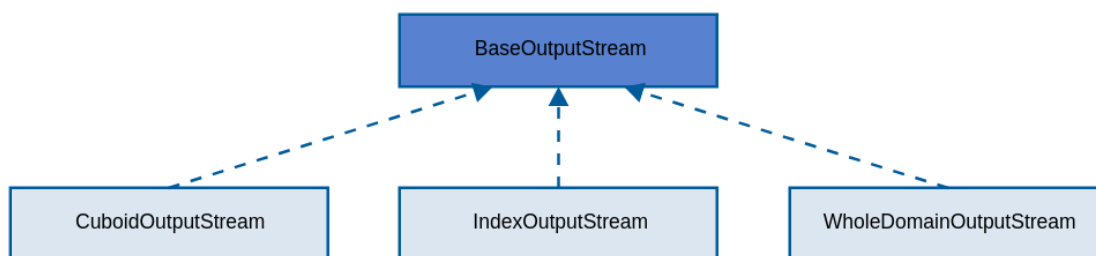
Trieda implementuje verejné metódy pre pridanie výstupného prúdu do kontajnera (alokácia pamäte), otvorenie/uzatvorenie prúdu, vzorkovanie dát, uvoľnenie pamäte, zistenie veľkosti kontajnera, atď. Ďalej implementuje chránenú metódu `createOutputStream`, ktorá

na základe parametrov vytvorí a vráti požadovaný výstupný prúd (objekt `BaseOutputStream`).

Triedy pre výstupné prúdy

Základnou triedou, z ktorej dedia všetky triedy výstupných prúdov je trieda `BaseOutputStream` (definuje rozhranie). Hierarchia tried je zobrazená na Obrázku 2.5. Rozhranie triedy tvoria metódy na vytvorenie prúdu, otvorenie/uzatvorenie prúdu, vzorkovanie dát, metóda na spracovanie dát. Implementované sú chránené metódy pre alokáciu/uvoľnenie pamäte. Ako členské premenné obsahuje typ redukčného operátora (`ReduceOperator`), ktorý bude použitý pri vzorkovaní dát, ďalej napr. objekt typu `Hdf5File` reprezentujúci výstupný súbor pre uloženie vzorkovaných dát.

Podľa typu vzorkovaných dát sú od triedy `BaseOutputStream` odvodené nasledovné triedy: `CuboidOutputStream` (vzorkuje dáta podľa masky tvaru kvádra určenom dvoma vrcholmi, pre každý senzor sa vytvorí samostatný set dát), `IndexOutputStream` (vzorkuje dáta podľa indexov masky senzorov) a `WholeDomainOutputStream` (vzorkuje dáta v celej doméne, ktoré sú uložené v jednom sete dát). Obrázok 2.5 zobrazuje hierarchiu dedičnosti tried výstupných prúdov.



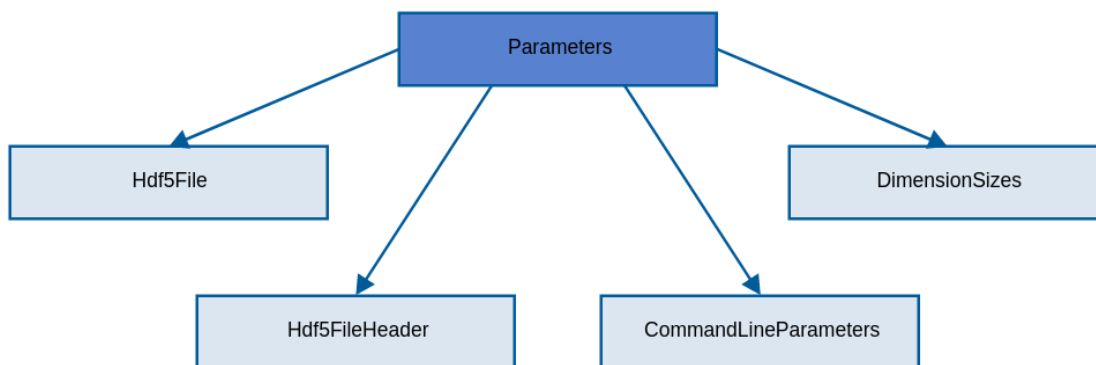
Obr. 2.5: Diagram dedičnosti tried výstupných prúdov.

Trieda Parameters

Trieda uchováva všetky parametre simulácie. Základné parametre programu sú najskôr načítané z prepínačov zadaných pri jeho spúšťaní (sú to napr. názov vstupného/výstupného súboru, počet *OpenMP* vlákien, rôzne príznaky pre ukladanie/vzorkovanie výstupných veličín apod.). Tieto parametre sú spracované a uložené triedou `CommandLineParameters`, ktorej objekt je súčasťou triedy `Parameters`. Po otvorení vstupného súboru sa postupne načítavajú hodnoty, ktoré definujú všetky vlastnosti simulácie. Sú to napr. veľkosti jednotlivých dimenzií domény, počet krokov simulácie, vlastnosti média (homogénne vs. heterogénne), linearita/nelinearita akustickej vlny, absorpcia média, vlastnosti zdrojov, tlak, rýchlosti, masky senzorov apod.

Trieda implementuje verejné metódy pre získanie každého z parametrov príkazového riadka alebo vstupného súboru. Trieda `Parameters` je v rámci celého programu inštanciovaná iba jedenkrát, jedná sa teda o návrhový vzor *Singleton*¹⁰. Na Obrázku 2.6 je zobrazený diagram tried, ktoré sú členmi triedy `Parameters` (zjednodušený diagram spolupráce tried, zobrazuje najvýznamnejšie triedy).

¹⁰<http://www.oodeesign.com/singleton-pattern.html>



Obr. 2.6: Diagram spolupráce pre triedu `Parameters`.

Triedy pre prácu so súbormi

Projekt k-Wave pracuje s vstupnými a výstupnými súbormi typu *HDF5*¹¹ (hierarchický dátový formát). Trieda `Hdf5File` obaľuje a zjednodušuje prácu práve s knižnicou *HDF5*. Sú tu implementované metódy na vytvorenie, otvorenie či uzatvorenie súboru, ďalej na vytvorenie, čítanie a modifikáciu obsahu súboru.

Samotné súbory obsahujú okrem vstupov/výstupov simulácie aj ďalšie informácie. Sú to napr. dátum vytvorenia súboru, popis súboru, verzia programu, spotreba pamäte, doba simulácie a mnohé iné. Tieto dodatočné informácie sú uložené v hlavičke *HDF5* súboru, ktorá je spracovávaná triedou `Hdf5FileHeader`. Ďalšie informácie k formátu *HDF5* sú uvedené v Odseku 5.3.

Pomocné triedy

Na informovanie užívateľa o stave simulácie, zaznamenávanie priebehu simulácie, prípadne o chybách, ktoré nastali počas behu programu slúži trieda `Logger`. Metóda `log` vypisuje na štandardný výstup požadované správy s rôznym stupňom dôležitosti. Na chybové hlásenia je implementovaná metóda `error`, prípadne metóda `errorAndTerminate`, ktorá vypíše chybu na štandardný chybový výstup a ukončí program. V rámci celého programu existuje jedna statická inštancia tejto triedy (*Singleton*).

Trieda (štruktúra) `DimensionSizes` je určená na uchovanie veľkostí jednotlivých dimenzií domény (načítané zo vstupného súboru). Trieda `TimeMeasure` slúži na meranie a záznam času spotrebovaného v jednotlivých častiach výpočtu.

2.2.3 Optimalizácia kódu

Pre dosiahnutie maximálneho výkonu je *C++* verzia optimalizovaná pre rôzne platformy. Jednotlivé verzie môžeme podľa pamäťového modelu rozdeliť nasledovne:

1. Využívajúce zdieľanú pamäť (UMA alebo NUMA)^[4]
 - *OpenMP*
 - procesory Intel x86

¹¹<https://support.hdfgroup.org/HDF5/>

- akcelerátor Intel Xeon Phi¹²
- určené pre klasické PC, pracovné stanice, jeden uzol superpočítača, ...
- jeden uzol superpočítača s 16 jadrami a 128 GB pamäte je vhodný pre simulácie do veľkosti 1024³
- *Nvidia CUDA*
 - akcelerátory Nvidia GPU ¹³, určené pre PC s akceleratorom Nvidia GPU
 - bežné GPU (Nvidia GTX 980) – simulácie do veľkosti 384³
 - architektúra Nvidia Pascal (až 24 GB pamäte) – simulácie do veľkosti 512³

2. Využívajúce distribuovanú pamäť

- *MPI*
 - zväzok procesorov Intel x86
 - zväzok akceleratorov Intel Xeon Phi
 - simulácie o veľkosti 1024³ a viac
- *MPI + OpenMP*
 - hybridný model
 - zväzok procesorov Intel x86
 - zväzok akceleratorov Intel Xeon Phi
- *MPI + Nvidia CUDA*
 - hybridný model
 - zväzok procesorov Intel x86 + akcelerátory Nvidia GPU

2.2.4 Implementácia k-Wave využívajúca zdieľanú pamäť

V nasledujúcej časti bude popísaná implementácia k-Wave pre stroje s architektúrou zdieľaného adresového priestora. Táto implementácia je akcelerovala pomocou *OpenMP* a jedná sa o verziu, ktorá bude použitá pre finálnu 2D a 3D implementáciu. Táto verzia vznikala postupne zo sekvenčného *C++* kódu. *OpenMP* je použité na optimalizáciu výpočtových jadier, ktoré operujú nad veľkým množstvom dát. V nasledujúcej časti budú použité rôzne prostriedky a pojmy *OpenMP*, ktoré sú stručne vysvetlené v Odseku 5.2.

Prvé použitie *OpenMP* v programe nastáva po alokácii pamäte jednotlivých matíc, kde sú matice paralelne inicializované nulami. Táto technika sa nazýva "*NUMA first touch*" a zabezpečuje vhodnú lokalitu dát. Kód inicializácie matíc je zobrazený vo Výpise 2.1. V ukážke sú použité direktívy *OpenMP* pre vytvorenie tímu vlákien (`parallel`), paralelizáciu cyklu (`for`) zo statickým plánovaním (`schedule`) a vektorizáciu cyklu `simd`.

```

1 // first touch
2 #pragma omp parallel for simd schedule(static)
3 for (size_t i = 0; i < mCapacity; i++)
4 {
5     mData[i] = 0.0f;
6 }

```

Výpis 2.1: Paralelná inicializácia matíc nulami (NUMA first touch)

¹²<https://www.intel.com/content/www/us/en/products/processors/xeon-phi/xeon-phi-processors.html>

¹³<https://developer.nvidia.com/cuda-gpus>

Typická implementácia výpočtového jadra akcelerovaného pomocou *OpenMP* je zobrazená vo Výpise 2.2. Použité direktívy sú podobné ako v predchádzajúcom príklade 2.1. Dôležitou vlastnosťou algoritmu je, že v cykloch sa iteruje cez "surové" ukazovatele, do ktorých sú priradené ukazovatele na matice z kontajnera *MatrixContainer*. Hodnoty jednotlivých prvkov matice sú teda modifikované mimo samotnú triedu. Tento spôsob bol zvolený z dôvodu vektorizácie najvnútornejšieho cyklu *for*. V dobe implementácie totiž kompilátor nepodporoval vektorizáciu nad polami zapúzdrenými v *C++* triedach. Vonkajší cyklus *for* je paralelizovaný pomocou direktív *OpenMP*. To znamená, že doména je v dimenzii *Z* rozdelená na subdomény (typicky v tvare kvádra), ktoré sú spracovávané paralelne skupinou vlákien (každé vlákno jedna subdoména). Bolo by samozrejme možné rozdeliť doménu aj v dimenzii *Y*. Tento spôsob by bol ale menej efektívny, pretože by jednotlivé vlákna spracovávali menšiu porciu dát a rozdeľovanie dát medzi vlákna by sa vykonávalo podstatne častejšie (veľká réžia).

```

1 // ...
2 const float* kappa = getKappa().getData();
3
4 FloatComplex* tempFftX = getTempFftwX().getComplexData();
5 FloatComplex* tempFftY = getTempFftwY().getComplexData();
6 FloatComplex* tempFftZ = getTempFftwZ().getComplexData();
7
8 FloatComplex* ddxKShiftNeg = getDdxKShiftNeg().getComplexData();
9 FloatComplex* ddyKShiftNeg = getDdyKShiftNeg().getComplexData();
10 FloatComplex* ddzKShiftNeg = getDdzKShiftNeg().getComplexData();
11
12 #pragma omp parallel for schedule(static)
13 for (size_t z = 0; z < reducedDimensionSizes.nz; z++)
14 {
15     for (size_t y = 0; y < reducedDimensionSizes.ny; y++)
16     {
17         #pragma omp simd
18         for (size_t x = 0; x < reducedDimensionSizes.nx; x++)
19         {
20             const size_t i = get1DIndex(z, y, x, reducedDimensionSizes);
21             const float eKappa = divider * kappa[i];
22
23             tempFftX[i] *= ddxKShiftNeg[x] * eKappa;
24             tempFftY[i] *= ddyKShiftNeg[y] * eKappa;
25             tempFftZ[i] *= ddzKShiftNeg[z] * eKappa;
26         } // x
27     } // y
28 } // z
29 // ...

```

Výpis 2.2: Paralelná implementácia 3D výpočtového jadra

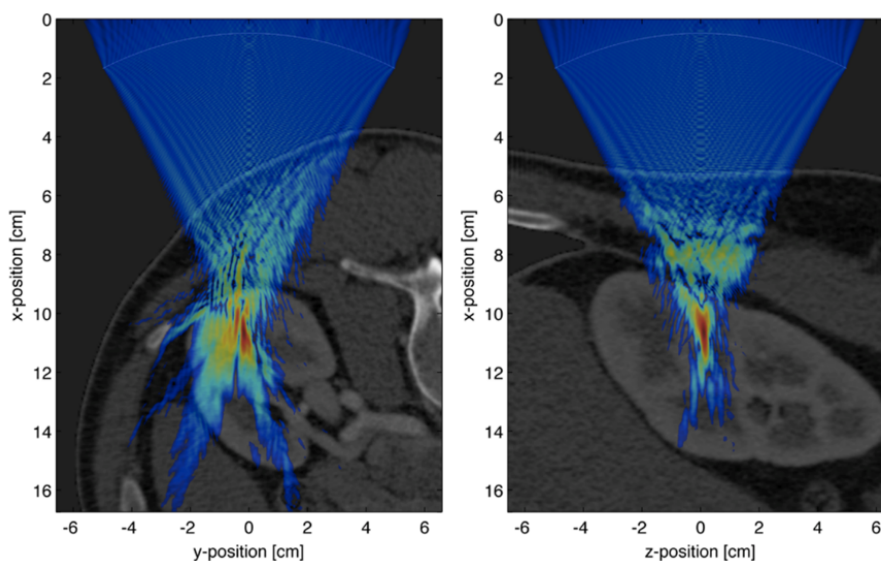
2.3 Príklad spustenia programu

Po kompilácii zdrojového kódu sa vytvorí spustiteľný súbor *kspaceFirstOrder3D-OMP*. Jedná sa o aplikáciu spustiteľnú v príkazovom riadku. Pri spúšťaní programu je možné zadať rôzne parametre definujúce jeho chovanie. Dostupné prepínače programu je možné zistiť jeho spustením s parametrom *-h* (nápoveda programu). Výpis všetkých prepínačov

programu je uvedený v prílohe **A**. Príklad spustenia programu pre výpočet simulácie nad konkrétnym vstupným súborom je uvedený v prílohe **B**. V tomto prípade sú parametrami programu vstupný súbor, výstupný súbor, počet vlákien programu a výstupná veličina, ktorá bude vzorkovaná.

2.4 Prípady použitia

Efektívna implementácia k-Wave pre superpočítače, ktorá je škálovateľná a distribuovaná rozšírila pole využiteľnosti programu. Hlavným zmyslom programu je simulácia ultrazvukových vln v heterogénnych tkanivách. Ako bolo spomenuté v Kapitole **1**, nástroj k-Wave je využiteľný na simuláciu ultrazvuku v ľudskom tele. V biofyzike a medicíne sú to napríklad metódy HIFU (operácia prostaty, obličiek), neuromodulácia a neurostimulácia. Ďalším praktickým použitím k-Wave je fotoakustické zobrazovanie [7], teda rekonštrukcia obrazu zo zachyteného signálu v čase a priestore. Táto rekonštrukcia je základným krokom pri fotoakustickom zobrazovaní pomocou tomografie. Pre tento prípad sa používa pôvodný model, ale v reverznom režime (algoritmus rekonštrukcie obrazu). Na Obrázku **2.7** môžeme vidieť ukážku simulácie ultrazvuku v obličke (HIFU) [15]. Obrázok zobrazuje maximálny akustický tlak v pozorovanej doméne (červené časti sú ohniskové body do ktorých je ultrazvuk smerovaný).



Obr. 2.7: Simulácia šírenia ultrazvuku v obličke [15].

Kapitola 3

Superpočítače

Rýchli rozvoj vedy a technológií kladie čoraz väčšie požiadavky na výpočtový výkon z hľadiska rýchlosti určitých operácií alebo množstva spracovaných dát. Superpočítače hrajú dôležitú rolu v modernej vede a sú používané pre riešenie širokej škály úloh, zaoberajúcimi sa napr. kvantovou mechanikou, predpoveďou počasia, modelovaním a simuláciou najrôznejších fyzikálnych či medicínskych problémov, kryptoanalýzou atď.

Do rozvoja a budovania superpočítačových centier sa v dnešnej dobe zapája mnoho krajín. Medzi najväčších gigantov v počte superpočítačov patria bezpochyby Čína a USA. Každá z krajín vlastní viac ako 170 superpočítačov umiestnených v rebríčku *TOP500*¹. Čína podľa údajov z júna 2017² vlastní najvýkonnejší superpočítač na svete (Sunway TaihuLight) s výkonom 93 PFLOPS. Medzi 500 najvýkonnejších superpočítačov sveta patrí aj Český superpočítač Salomon³.

3.1 Architektúra superpočítača

Superpočítač sa skladá z veľkého množstva procesorov a disponuje vysokým výkonom. Na jednej základnej doske sa nachádza niekoľko procesorov, spolu s operačnou pamäťou RAM, sieťovou kartou, prípadne akcelarátorami rôzneho druhu. Tieto komponenty spolu tvoria 1 uzol superpočítača. Vysoký výkon je dosiahnutý jednak pomocou špičkového hardware samotného uzla, ale najmä pomocou prepojenia uzlov do zväzku (angl. cluster).

Základným ukazovateľom výkonu superpočítača je počet "floating point" operácií vykonaných za 1 sekundu (angl. *FLOPS* – Floating Point Operations Per Second). Dnešné superpočítače sa pohybujú v ráde desiatok *PetaFLOPS*. Ďalšími dôležitými parametrami sú priepustnosť pamäťového subsystému (*GB/s*), priepustnosť (sieťového) prepojenia medzi uzlami (*GB/s*), priepustnosť I/O (rýchlosť čítania/zápisu na disk *GB/s*), počet CPU jadier, veľkosť pamäte RAM, atď.

3.1.1 Architektúra jedného uzla

Uzol superpočítača tvorí serverová základná doska, na ktorej bývajú osadené zvyčajne dva procesory (dva sockety). Procesory sú na základnej doske medzi sebou prepojené rýchlosťou zbernicou alebo linkami (závisí na konkrétnej architektúre). Napríklad procesory Intel sú

¹<https://www.top500.org>

²<https://www.top500.org/lists/2017/06/>

³<https://docs.it4i.cz/salomon/introduction/>

medzi sebou prepojené pomocou rýchlych liniek QPI⁴ (Quick Path Interconnect). Každý procesor má niekoľko jadier (napr. 8–16), ktoré medzi sebou zdieľajú cache poslednej úrovne (LLC)⁵. V procesore je integrovaný pamäťový radič (môže ich byť viac), ku ktorému je pripojená pamäť RAM (napr. 128 GB).

Pre komunikáciu s okolím je uzol vybavený sieťovou kartou. Kvôli nárokom na rýchlu komunikáciu s ostatnými uzlami sa jedná o špeciálny typ karty, ktorá zabezpečuje veľkú priepustnosť a malú latenciu. Uzol môže obsahovať aj lokálny disk (úložisko dát), ale v dnešných superpočítačoch sa používajú skôr centralizované diskové polia (rôzne úrovne), do ktorých uzol prístupuje po sieti. Využíva sa teda sieťový, paralelný súborový systém, napr. *Lustre*⁶. Väčšina superpočítačov beží na operačnom systéme Linux/Unix.

Uzol superpočítača môže byť vybavený akceleračnou kartou, ktorá slúži na urýchlenie výpočtu určitých typov úloh. V dnešných superpočítačoch sa používajú najmä akcelerátory Nvidia GPU a Intel Xeon Phi. Akcelerátory sú do systému pripojené typicky pomocou rozhrania PCI Express.

3.1.2 Prepojenie uzlov

Prepojenie jednotlivých komponent superpočítača je realizované pomocou prepojovacej siete. Táto sieť pozostáva z prepojovacích liniek a prepínačov, ktoré sú usporiadané do určitej topológie. Podľa usporiadania a funkcie jednotlivých komponent rozdeľujeme siete na priame a nepriame [4]. Priama sieť pozostáva z niekoľkých liniek typu "point-to-point". Každý uzol je súčasne výpočtovým aj prepínacím (distribúované prepínače, rovnocenné uzly). Zástupcami priamej siete sú napríklad rôzne druhy hyperkocky, torus apod. Nepriama sieť pozostáva z prepojovacích liniek a nerovnocenných uzlov, ktoré sú buď výpočtové, alebo prepínacie (centralizované prepínače). Príkladom tejto siete je sieť "Closs", alebo "Fat tree".

Na fyzické prepojenie komponent superpočítača sa používajú špeciálne technológie, vyvinuté práve pre tieto účely. Prepojovacia sieť rieši viacero funkcií, ktoré sa odohrávajú užívateľovi na popredí (sprístupnenie superpočítača užívateľom – ssh, VNC), ale aj na pozadí (komunikácia uzlov medzi sebou, sieťový súborový systém, ...). Často používanými technológiami sú Ethernet a InfiniBand⁷ (používa sa aj kombinácia oboch). Komunikácia uzlov na úrovni aplikácií prebieha pomocou rozhrania pre zasielania správ (angl. Message Passing Interface – *MPI*).

3.2 Plánovanie úloh

Na superpočítači pracuje zvyčajne veľké množstvo užívateľov, preto je nutné dômyselné plánovanie úloh, ktoré majú byť spustené. Superpočítač môže navyše obsahovať rôzne typy uzlov, kde na každý typ môže byť viazaná iná politika pridelovania prostriedkov, účtovania procesorového času apod. Podľa typu bývajú uzly zvyčajne usporiadané do takzvaných front. Je teda potrebný systém, ktorý rieši plánovanie úloh, alokáciu výpočtových prostriedkov, spúšťanie úloh a reportovanie výsledkov užívateľovi. K týmto účelom slúži práve systém pre dávkové spracovanie úloh – *PBS*⁸ (angl. Portable Batch System).

⁴<https://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html>

⁵LLC – Last Level Cache

⁶<http://lustre.org/>

⁷https://www.mellanox.com/pdf/whitepapers/IB_Intro_WP_190.pdf

⁸<http://pbspro.org/>

Užívateľ má zvyčajne možnosť spustenia interaktívnej alebo dávkovej úlohy. U interaktívneho módu dostane užívateľ po pridelení prostriedkov priamy prístup (napr. pomocou ssh, vnc) na daný uzol, prípadne viac uzlov. U dávkového módu užívateľ pripraví takzvaný *PBS* skript⁹, ktorým len zaradí úlohu do fronty a po skončení dostane notifikáciu o ukončení výpočtu.

PBS skript musí presne definovať požadované prostriedky (typ fronty, výpočtový čas, počet uzlov, počet jadier, veľkosť RAM). Úloha je zaradená do príslušnej fronty po spustení príkazu `qsub` pre daný *PBS* skript. Po uvoľnení požadovaných prostriedkov je úloha spustená na cieľovom stroji.

```
1 !/bin/bash
2 #PBS -A OPEN-6-11
3 #PBS -q qprod
4 #PBS -l walltime=10:00:00
5 #PBS -l select=1:ncpus=24
6 #PBS -e error.txt
7
8 cd $PBS_O_WORKDIR
9 ml HDF5/1.8.18-intel-2017a-serial intel/2017.00
10 export OMP_NUM_THREADS=12
11
12 ./omp-example
```

Výpis 3.1: Príklad *PBS* skriptu

Vo Výpise 3.1 je uvedený príklad jednoduchého *PBS* skriptu. Prvý riadok určuje interpret skriptu, druhý názov projektu, do ktorého bude započítaný spotrebovaný procesorový čas. Parameter `-q` určuje názov fronty, do ktorej bude úloha zaradená. Parameter `-l` určuje požadované prostriedky, v našom prípade 10 hodín času pre výpočet, 1 uzol, 24 jadier. Parameter `-e` určuje súbor, do ktorého bude zapísané prípadné chybové hlásenie. Ďalej nasleduje prepnutie pracovného adresára, načítanie požadovaných modulov, nastavenie premenných prostredia a spustenie samotného programu.

3.3 Superpočítač Salomon

Počas tvorby tejto práce boli využívané prostriedky superpočítačov Anselm¹⁰ a Salomon¹¹, ktoré sú umiestnené v národnom superpočítačovom centre českej republiky *IT4Innovations*¹². Nižšie bude popísaný práve superpočítač Salomon, ktorý ešte aj na konci roka 2017 patril medzi 500 najvýkonnejších superpočítačov sveta (87 miesto¹³).

Cluster Salomon [6] sa skladá z 1008 výpočtových uzlov, ktoré spolu disponujú 24192 CPU jadrami a 129 TB pamäte RAM. Maximálny teoretický výkon superpočítača je viac ako 2 PFLOPS. Každý výpočtový uzol je výkonný počítač s 24 jadrami architektúry x86-64 a minimálne 128 GB pamäte RAM. Uzle sú navzájom prepojené pomocou rozšírenej 7D hyperkocky¹⁴. Fyzické prepojenie uzlov je realizované vysoko rýchlostnou sieťou InfiniBand a Ethernet.

⁹<http://www.pbsworks.com/pdfs/PBSUserGuide13.0.pdf>

¹⁰<https://docs.it4i.cz/anselm/introduction/>

¹¹<https://docs.it4i.cz/salomon/introduction/>

¹²<http://www.it4i.cz/>

¹³<https://www.top500.org/list/2017/11/>

¹⁴<https://docs.it4i.cz/salomon/7d-enhanced-hypercube/>

Cluster sa skladá z 576 uzlov bez akcelerátora a 432 uzlov s akcelerátorom Intel Xeon Phi. Na superpočítači beží linuxový operačný systém CentOS. Všetky uzle zdieľajú 0.5 PB diskové úložisko určené pre domovské adresáre užívateľov (súborový systém NFS). Okrem toho sú dostupné aj zdieľané úložiská (pre dočasné aj trvalé dáta) s veľkou kapacitou (súborový systém Lustre).

Prístup na superpočítač zabezpečujú 4 uzle (`login[1-4]`), ktoré sú vyhradené na prihlasovanie užívateľov, tvorbu úloh a iné, výpočtovo nenáročné prípady. Nakoľko sa jedná o linuxový operačný systém, prístup je možný pomocou protokolu *ssh* (textový mód, príkazový riadok). Základné grafické rozhranie aplikácií (tunelovanie jednotlivých aplikácií pomocou *ssh*) je sprístupnené pomocou systému *X window*¹⁵. Pre plnohodnotné grafické užívateľské rozhranie je možné využiť protokol *VNC*¹⁶.

3.3.1 Modulárny systém

Nakoľko na superpočítači pracuje veľké množstvo užívateľov zameraných na rôzne oblasti a technológie, je nutná dostupnosť veľkého množstva knižníc, programov a nástrojov. Všetky knižnice, programy a nástroje sú teda členené do takzvaných modulov. Jednotlivé moduly sú podľa zamerania delené do určitej štruktúry. Sú to napríklad moduly pre bioinformatiku, chémiu, matematiku, prekladače, debugery, knižnice apod.

Pre prácu s modulmi (výpis dostupných modulov, načítanie/odstránenie modulu ...) sa používa príkaz `module`, prípadne kratšia alternatíva `m1`. Vo Výpise 3.2 je znázornený príklad práce s príkazom `module` (načítanie modulu *OpenMPI*). Ako môžeme vidieť, *OpenMPI* je závislý na ďalších moduloch, ktoré sú automaticky použité pri načítavaní požadovaného modulu.

```
1 [xsimek23@login1.salomon ~]$ module list
2 No modules loaded
3 [xsimek23@login1.salomon ~]$ module load OpenMPI
4 [xsimek23@login1.salomon ~]$ module list
5
6 Currently Loaded Modules:
7 1) GCCcore/7.1.0                4) numactl/2.0.11-GCC-7.1.0-2.28 (H)
8 2) binutils/2.28-GCCcore-7.1.0  5) hwloc/1.11.7-GCC-7.1.0-2.28
9 3) GCC/7.1.0-2.28              6) OpenMPI/1.10.7-GCC-7.1.0-2.28
10
11 [xsimek23@login1.salomon ~]$ module unload OpenMPI
```

Výpis 3.2: Príklad načítania modulu *OpenMPI*, zobrazenie načítaných modulov, odstránenie načítaného modulu

3.3.2 Výpočtové uzly

Salomon je zložený z uzlov architektúry Intel x86-64, ktoré sú spravované (alokácia prostriedkov, pridelovanie úloh) systémom *PBS* (Odsek 3.2). Jednotlivé uzly sú zaradené do takzvaných front podľa spôsobu využitia (napr. expresné, produkčné, vizualizačné fronty a pod.). Salomon sa skladá z nasledujúcich typov uzlov:

¹⁵<http://www.opengroup.org/desktop/x/>

¹⁶<https://www.realvnc.com/en/>

Výpočtové uzly bez akcelarátorá

- 576 uzlov, spolu 13824 jadier CPU
- každý uzol osadený 2x procesorom Intel Xeon E5-2680v3
 - 12 jadier, 2.5 GHz (3.3 GHz Turbo Boost)
 - maximálny výkon 19.2 GFLOPS na 1 jadro
 - 30 MB Intel Smart Cache, priepustnosť 68 GB/s
- 128 GB pamäť RAM
 - 8x DDR4 DIMM na 1 uzol, 4x DDR4 DIMM na 1 procesor, 1x DDR4 DIMM na 1 pamäťový kanál
 - obsadená pamäť 8x 16 GB DDR4 DIMM 2133 MHz

Výpočtové uzly s akcelarátorom

- 432 uzlov, spolu 10368 jadier CPU
- procesory a pamäť rovnaké, ako v uzloch bez akcelarátorá
- 2x MIC¹⁷ Intel Xeon Phi 7120P
 - 61 jadier, 1.238 GHz (1.333 GHz Turbo Boost)
 - maximálny výkon 18.4 GFLOPS na 1 jadro
 - 30.5 MB L2 cache, priepustnosť 352 GB/s
 - 16 GB pamäť RAM
 - * 2x socket
 - * 16x GDDR5 DIMM na 1 uzol, 8x GDDR5 DIMM na 1 procesor, 2x DDR4 DIMM na 1 pamäťový kanál

Uv 2000

- 1 uzol, spolu 112 jadier CPU
- 14x procesor Intel Xeon E5-4627v2, 14 NUMA uzlov
 - 8 jadier, 3.3 GHz (3.6 GHz Turbo Boost)
 - 16 MB Intel Smart Cache, priepustnosť 59.7 GB/s
- 3328 GB pamäť RAM
- 1x NVIDIA GM200 (GeForce GTX TITAN X)
 - 3072 CUDA jadier, 1000 MHz (1075 MHz Boost Clock)
 - priepustnosť pamäte 336.5 GB/s
 - 12 GB pamäť RAM, GDDR5

¹⁷MIC – Many Integrated Core

Kapitola 4

Paralelný výpočet

Paralelný výpočet [3], [4] je dnes jedinou rozumnou možnosťou pre dosiahnutie podstatne vyššej výkonnosti aplikácií. Pred 15–20 rokmi stačilo počkať cca jeden rok, aby program bežal rýchlejšie na novej generácii procesorov. Zvyšovanie výkonnosti pomocou zvyšovania frekvencie jednojadrových procesorov sa zastavilo okolo roku 2003, kvôli dosiahnutiu technologických možností. Novou možnosťou zvyšovania výkonu procesorov sa v tejto dobe stal práve paralelný výpočet (viac jadrové CPU).

Paralelné systémy sa dajú rozdeliť podľa niekoľkých klasifikácií. Tradičnou klasifikáciu paralelných systémov je *Flynnova* [4] klasifikácia:

1. SISD

- Single Instruction, Single Data
- sekvenčný počítač

2. SIMD

- Single Instruction, Multiple Data
- vektorové a maticové procesory
- multimediálne rozšírenia (SSE, AVX)
- GPU SIMT (Single Instruction, Multiple Threads)

3. MIMD

- Multiple Instruction, Multiple Data
- pre jednoduchosť sa často programujú podľa vzoru SPMD (Single Program, Multiple Data) s podmienenými časťami

4. MISD

- Multiple Instruction, Single Data
- teoretický model, komerčná implementácia neexistuje

Ďalej rozlišujeme 3 paralelné (abstraktné) programovacie modeli, resp. 3 architektúry strojov: zdieľaný adresový priestor, zasielanie správ a dátovo paralelný model.

4.1 Zdieľaný adresový priestor

Zdieľaný adresový priestor (angl. SAS – Shared Address Space) zodpovedá MIMD vo Flynnovej klasifikácii

(SPMD). Jedná sa o prirodzené rozšírenie sekvenčného programovacieho modelu, paralelizmus na úrovni vlákien. Vlákna komunikujú čítaním/zápisom do zdieľaných premenných. Komunikácia je implicitná v pamäťových operáciach (jednoduché na programovanie, ale vyžaduje synchronizáciu). Každé vlákno môže čítať alebo zapisovať do ľubovolnej zdieľanej premennej (problém nastáva pri súčasnom čítaní a zápise).

Čo sa týka prístupu do pamäte, v rámci zdieľaného adresového priestoru rozlišujeme 2 architektúry:

1. Uniformný prístup do pamäte

- UMA – Uniform memory access
- všetky CPU môžu pristupovať na ľubovольné miesta v pamäti
- doba prístupu k dátam do pamäte, ktoré nie sú v lokálnej cache, je rovnaká pre všetky CPU
- s rastúcim počtom CPU je obtiažne zachovať rovnaký prístup do pamäte pre všetky CPU (rôzne vzdialenosti CPU od pamäte)

2. Neuniformný prístup do pamäte

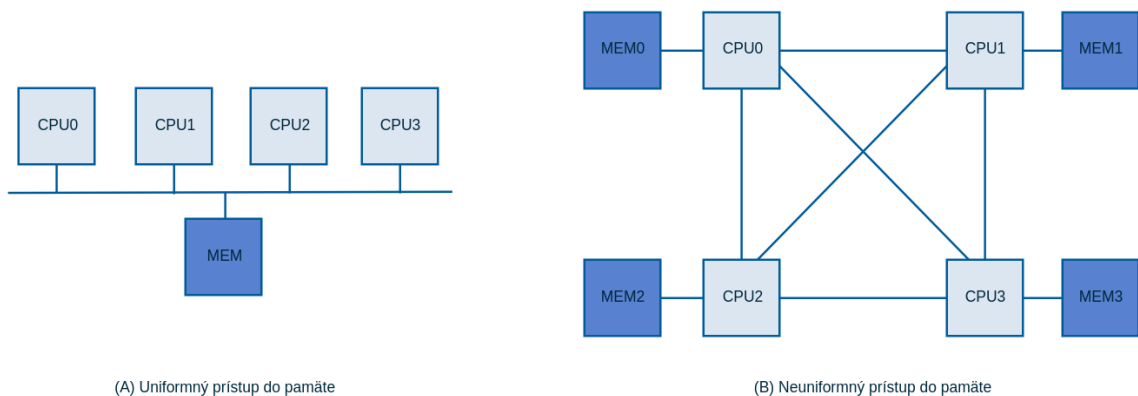
- NUMA – Non Uniform memory access
- všetky CPU môžu pristupovať na ľubovольné miesta v pamäti, ale rôzne CPU za rozdielnu dobu
- dá sa lepšie škálovať (väčší počet CPU)
- pri programovaní je ale nutné zohľadňovať lokalitu dát

Na Obrázku 4.1 sú znázornené jednotlivé architektúry. Na ľavo (A) je architektúra UMA. Jadrá *CPU0 – CPU3* sú k pamäti *MEM* pripojené pomocou spoločnej zbernice, každé jadro má rovnako dlhú dobu prístupu do pamäte. Na pravo (B) je architektúra NUMA. Je možné vidieť, že jadrá *CPU0 – CPU3* sú rôzne vzdialené od jednotlivých pamätí *MEM0 – MEM3*. Napr. jadro *CPU0* je pripojené priamou linkou k pamäti *MEM0*, ale do pamäte *MEM3* musí pristupovať cez ďalšiu 1 linku navyše. To znamená, že doba prístupu do jednotlivých pamätí je pre každé jadro rôzna.

Príkladom špičkového systému SAS – NUMA je uzol *Uv 2000* v Ostravskom superpočítaní Salomon (14 procesorov, 112 jadier, 3.3TB RAM). Príkladom HPC jazyka slúžiaceho na akceleráciu programov pre systémy zo zdieľanou pamäťou (UMA aj NUMA) je *OpenMP*.

4.2 Zasielanie správ

Zasielanie správ (MP – Message Passing) zodpovedá MIMD vo Flynnovej klasifikácii, (SPMD). Vlákna (procesy) pracujú v rámci nezávislých adresových priestorov, na každom procesore (jadre) beží nezávislý proces. Dáta nie sú medzi procesormi zdieľané, ale sú explicitne kopírované. Procesy medzi sebou komunikujú zasielaním správ cez prepojavaciu sieť. Synchronizácia procesov prebieha buď pri dátovej komunikácii (prenos dát) alebo oddelene pomocou zasielania synchronizačných správ.



Obr. 4.1: UMA a NUMA architektúra.

Abstraktný model zasielania správ je popísaný štandardom *MPI*¹. Samotné *MPI* teda nie je jazyk, ale štandard, resp. špecifikácia chovania funkcií rozhrania pre zasielanie správ. Medzi implementácie *MPI* patrí napr. *MPICH*², *Microsoft MPI*³, *Intel MPI*⁴, *OpenMPI*⁵. Cieľom *MPI* je prepojenie úplných systémov (PC, serverov) do jedného, funkčného celku (zväzok – cluster). Príkladom veľkého systému, v ktorom medzi sebou uzly komunikujú zasielaním správ je superpočítač Salomon.

Zasielanie správ je v porovnaní so zdieľaným adresovým priestorom náročnejšie na programovanie, pretože vyžaduje explicitné príkazy na zdieľanie dát a synchronizáciu. U modelu SAS je možné program paralelizovať inkrementálne (začneme postupne od kritických miest). Naopak model MP núti uvažovať paralelné spracovanie hneď od začiatku návrhu a tvorby programu. Na druhej strane, model MP je jednoducho rozšíriteľný na veľký počet procesorov či uzlov, zatiaľ čo SAS je obmedzený na fungovanie v rámci 1 uzla. Veľmi často sa používa takzvaný hybridný model, ktorý je kombináciou oboch vyššie uvedených modelov. Na komunikáciu medzi uzlami sa používa zasielanie správ, zatiaľ čo v rámci 1 uzla funguje SAS.

4.3 Dátovo paralelný model

Dátovo paralelný model zodpovedá SIMD vo Flynnovej klasifikácii (SIMT). Vykonáva sa rovnaká operácia (inštrukcia) na každom prvku vektora. V moderných procesoroch sa jedná napr. o multimediálne rozšírenia SSE či AVX.

Iný pohľad na dátovo paralelný model môže byť programovanie podľa vzoru SPMD, kde sa na kolekciu dát mapuje určitá funkcia (`map(function, collection)`). Kolekciou dát môže byť v tomto prípade vektor, sada súborov, prúd dát apod. Podmienkou je, že aplikácia danej funkcie je pre každý prvok kolekcie nezávislá. To znamená, že neprebíha žiadna komunikácia počas mapovania funkcie na kolekciu (prípadne veľmi obmedzená komunikácia/synchronizácia). Synchronizácia je implicitná až po ukončení mapovania.

¹<https://www.mpi-forum.org/docs/>

²<https://www.mpich.org/>

³[https://msdn.microsoft.com/en-us/library/bb524831\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/bb524831(v=vs.85).aspx)

⁴<https://software.intel.com/en-us/intel-mpi-library>

⁵<https://www.open-mpi.org/>

Ďalším súvisiacim modelom je programovací model pracujúci s prúdom dát. Dátový paralelizmus je v tomto prípade vyjadrený ako operácie (výpočtové jadrá, angl. compute kernels) vykonávané na prúde dát (časové rady). Využívajú sa procesory s veľkým množstvom výpočtových jednotiek (napr. floating point ALU na GPU). Nie je nutná explicitná alokácia, synchronizácia či komunikácia medzi výpočtovými jednotkami. Aplikácie tohto programovacieho modelu môžeme nájsť v spracovaní audia, videa, fyzikálne simulácie, kryptografia, počítačová grafika a v mnoho iných. Príkladmi akcelerátorov pracujúcich na tomto princípe sú Intel Xeon Phi (SIMD) či Nvidia GPU (SIMT).

4.4 Škálovanie

Škálovanie programu (algoritmu) je schopnosť využitia väčšieho množstva poskytnutých prostriedkov za účelom zvýšenia výkonnosti. Je veľmi dôležitým parametrom popisujúcim vlastnosti paralelného programu (algoritmu). V ideálnom prípade klesá doba výpočtu lineárne zo vzrastajúcim počtom procesorov (ak zvýšime počet procesorov na dvojnásobok, čas výpočtu sa zníži na polovicu). Táto situácia je ale vo väčšine programov nereálna, pretože škálovanie algoritmu má isté obmedzenia.

Rozlišujeme dva druhy škálovania. Prvé z nich je silné škálovanie (popisuje Amdahlov zákon [4], [14]), ktoré hovorí že nie je možné nekonečné zvyšovanie výpočtových jednotiek za účelom lineárneho zvyšovania výkonnosti. Tento fakt plynie z obmedzenia, že nie všetky časti programu je možné paralelizovať. Amdahlov zákon hovorí, že ak je možné paralelizovať napr. 90% úlohy (10% úlohy musí bežať sekvenčne), tak nie je možné danú úlohu zrýchliť viac ako 10 krát. Ak by sme za použitia nekonečného množstva procesorov znížili čas paralelnej časti na 0, čas sekvenčnej časti tvorí stále 10% z pôvodnej doby výpočtu.

Slabé škálovanie (popisuje Gustafsonov zákon) zohľadňuje celkový výkon, ktorý získame zvýšením počtu výpočtových jednotiek, čo umožňuje riešiť väčší problém v rovnakom čase. Amdahlov zákon predpokladá, že veľkosť riešeného problému a veľkosť sekvenčnej časti je nezávislá na počte procesorov. Gustafson naopak predpokladá, že zo zvýšením počtu procesorov dôjde aj k nárastu riešeného problému (relatívne klesá sekvenčná časť), čo má za následok vyriešenie väčšieho problému za čas rovnaký ako vyriešenie pôvodného (menšieho) problému.

Kapitola 5

Prostriedky pre implementáciu

5.1 Prostriedky C++

Jazyk *C++* [11] prešiel od svojho počiatku mnohými zmenami a vylepšeniami. Bjarne Stroustrup položil základy *C++* v 80. rokoch, kedy jazyk *C* rozšíril o možnosti objektovo orientovaného programovania (OOP), teda pridal podporu objektov a tried. K rozšírenému jazyku vytvoril prekladač transformujúci jazyk *C++* na jazyk *C*, ktorý bol následne preložený do binárneho kódu pomocou dostupných kompilátorov. Postupne sa začali pridávať ďalšie vylepšenia, ako napr. operátor inkrementácie (z toho bol práve odvodený názov *C++*), výnimky a mnoho iného. Postupne tak vznikol nový, samostatný jazyk, ktorý je vyvíjaný a vylepšovaný dodnes. Cieľom tejto práce nie je podrobne popísať jazyk *C++*, nakoľko má obrovské množstvo možností, vlastností a funkcií. V nasledujúcej časti budú uvedené len prostriedky *C++*, ktoré sú z hľadiska tejto práce zaujímavé. Zaujímajú nás najmä koncepty využiteľné pre zjednotenie kódu 2D a 3D k-Wave.

5.1.1 Dedičnosť

Jedným z hlavných cieľov OOP je vytvorenie znovu použiteľného kódu. K tomuto cieľu sa hodí práve jeden z najdôležitejších konceptov jazyka *C++*, ktorým je dedičnosť tried [9]. Zjednodušene povedané, dedičnosť dovoľuje vytvoriť z takzvanej základnej triedy triedu odvodenú. Odvodená trieda dedí vlastnosti základnej triedy, pričom môže priamo pristupovať k členským prvkom základnej triedy, ktoré nie sú privátne. K privátnym členom základnej triedy môže odvodená trieda pristupovať pomocou metód základnej triedy, ktoré nie sú privátne. Odvodená trieda dedí všetky metódy základnej triedy, až na nasledujúce výnimky:

- konštruktor, deštruktor a kopírovací konštruktor základnej triedy
- preťažené operátory základnej triedy
- spriatelené metódy základnej triedy

Sila dedičnosti vychádza z možnosti rozšírenia vlastností základnej triedy o nové vlastnosti bez potreby modifikácie kódu základnej triedy. Odvodená trieda dedí funkcionality základnej triedy, pričom ju môže neobmedzene rozširovať a vylepšovať.

Dedičnosť sa teda javí ako potenciálny kandidát pre riešenie problému zjednotenia 2D a 3D kódu k-Wave. Jedným z návrhov bola možnosť implementácie 2D simulácie, ktorá by bola základom pre ďalšie rozšírenia. Nové 3D triedy by potom mohli dediť vlastnosti 2D tried

a zároveň pridať prvky potrebné pre 3D simuláciu. Súčasná podoba kódu by pravdepodobne umožňovala použitie tohto konceptu z hľadiska dát používaných v simulácii (kontajner matic). Kontajner pre 2D kód by obsahoval matice potrebné pre výpočty v dimenziách X a Y . Odvodený 3D kontajner by následne pridával matice potrebné pre výpočty v dimenzii Z .

Problém ale nastáva u metód simulačnej triedy, ktoré zabezpečujú výpočet jednotlivých krokov simulácie (výpočtové jadrá). Vo Výpise 2.2 je znázornený príklad 3D výpočtového jadra. Implementácia 2D výpočtového jadra je veľmi podobná, pričom vynecháva výpočty nad dimenziou Z . V niektorých metódach navyše zavádza pri výpočtoch rôzne konštanty odlišné od verzie 3D.

Zásadným faktom z hľadiska výkonu je umiestnenie direktívy pre paralelizáciu cyklu (jej význam bol popísaný v Odseku 2.2.4). V 3D kóde je nutné paralelizovať cyklus iterujúci cez dimenziu Z , pričom v 2D kóde cyklus iterujúci cez dimenziu Y . Tieto malé, ale zásadné zmeny by viedli na vetvenie výpočtových jadier na verziu 2D a 3D (veľký počet podmienok v jadrách), ktoré by neboli optimálne z hľadiska výkonu (podmienené príkazy v paralelných cykloch) ani z hľadiska štruktúry kódu.

Ďalšou možnosťou by bolo v odvodenej triede nahradiť 2D výpočtové jadrá za nadradené 3D jadrá ("overriding"). Toto riešenie by bolo optimálne z hľadiska výkonu, ale nie z hľadiska štruktúry kódu. Existovalo by viac verzií výpočtových jadier, čo by malo za následok ťažšiu údržbu a rozšíriteľnosť kódu. Pre novú implementáciu je požadovaný spôsob, ktorý by v ideálnom prípade zachoval rozumnú štruktúru kódu (jednoduchá údržba, rozšíriteľnosť, prehľadnosť, čitateľnosť) a optimálny výkon, ktorý by nebol degradovaný rozhodovaním sa za behu programu.

5.1.2 Generické programovanie

Spoločným cieľom generického programovania a objektovo orientovaného programovania je abstrakcia, a vytvorenie znovu použiteľného kódu. Ich filozofia sa však úplne líši. Zatiaľ čo objektovo orientované programovanie sa sústreďuje na dátový aspekt, generické programovanie sa sústreďuje na algoritmus.

Cieľom generického programovania [19] je vytvorenie kódu nezávislého na dátových typoch. Nástrojom jazyka *C++* pre vytváranie generických programov sú šablóny (angl. *templates*). Umožňujú nám definovať funkciu či dokonca triedu podľa generického typu. V jazyku *C++* existuje štandardná knižnica šablón *STL* (Standard Template Library), ktorá ponúka generickú reprezentáciu algoritmov.

Konkrétnym príkladom šablóny implementovanej v *STL* je takzvaný *kontajner*. Kontajner je objekt, v ktorom sú uložené iné objekty jednoduchého typu. Uložené objekty môžu byť objekty v zmysle OOP alebo hodnoty vstavaných typov (objekt musí byť kopírovateľný a priraditeľný). Dáta uložené v kontajnery sú jeho vlastníctvom, čo znamená, že pri zániku kontajnera typicky zanikajú aj objekty v ňom uložené. Príkladom kontajnera môže byť vektor (`std::vector<T>`), kde T je typ objektu uloženom vo vektore. Napríklad vektor obsahujúci celé čísla vytvoríme nasledovne:

```
std::vector<int> vec;
```

Pre iterovanie prvkami kontajnera sa používajú takzvané *iterátory*. Rovnako ako šablóny činia algoritmus nezávislý na type uložených dát, iterátory činia algoritmus nezávislý na type použitého kontajnera.

Výhodami *C++* šablón sú teda najmä možnosť generalizácie typov, zníženie redundancie kódu, vytváranie bezpečného kódu z hľadiska typov. Z nášho pohľadu je najzásadnejšou výhodou to, že šablóny sú pre konkrétny program generované pre konkrétne typy v dobe kompilácie. V zdrojovom kóde je použitie šablón veľmi kompaktné a neredundantné. V kompilátorom vygenerovanom binárnom kóde sa však nachádza kód pre každý, konkrétne použitý typ. To znamená, že v dobe prekladu sa program nemusí rozhodovať, aký typ objektu použiť, ktorú vetvu programu zvoliť atď. Šablóny je teda možné použiť aj na parametrizovanie funkcií či metód, u ktorých je vysoký výkon zásadnou požiadavkou.

Vo Výpise 5.1 je znázornený kód metódy, ktorá je parametrizovaná pomocou šablóny (`template<bool isScalarFlag>`). Funkcia pracuje buď s jedinou, skalárnou hodnotou alebo s maticou hodnôt. Kompilátor v dobe prekladu vytvorí obe verzie funkcie (pre skalárnu hodnotu aj pre maticu) a pri volaní funkcie sa použije jej konkrétna varianta. Ak by bola funkcia parametrizovaná klasickým parametrom funkcie (`void funcExample(bool isScalarFlag)`), tak by sa pri každom vykonaní funkcie, v každej iterácii cyklu musela vyhodnocovať podmienka, ktorá by zisťovala o aký typ sa práve jedná. Riešením z hľadiska výkonu by bolo naprogramovanie dvoch verzií funkcie, ktoré by boli totožné čo sa algoritmu týka, avšak pracovali by nad rôznym typom dát. Tento spôsob ale pridáva do zdrojového kódu značnú redundanciu. Šablóny nám teda dovoľujú parametrizáciu funkcií pre rôzne typy dát zo zachovaním neredundantného a výkonného kódu.

```

1 // parameter isScalarFlag určuje, či bude funkcia pracovať
2 // so skalárnou hodnotou alebo s maticou
3 template<bool isScalarFlag>
4 void funcExample(void)
5 {
6     // ...
7     // inicializácia skalárnej hodnoty a ukazovateľa matice podľa parametru isScalarFlag
8     const float scalar = isScalarFlag ? getScalar() : 0;
9     const float* matrix = isScalarFlag ? nullptr : getMatrix();
10    for (size_t i = 0; i < nElements; i++)
11    {
12        const float value = isScalarFlag ? scalar : matrix[i];
13        // použite premennej value ďalej vo výpočte
14        // ...
15    }
16    // ...
17 }
18
19 // volanie funkcie pre skalárny typ
20 funcExample<true>();
21 // volanie funkcie pre maticu
22 funcExample<false>();

```

Výpis 5.1: Príklad funkcie parametrizovanej pomocou šablóny

5.2 OpenMP

OpenMP (Open Multi – Processing) [2], [10] je aplikačné rozhranie (API) vytvorené konzorciom hlavných výrobcov hardware a software. *OpenMP* ponúka prenositeľný a škálovateľný model pre paralelné výpočty založené na programovaní so zdieľanou pamäťou. Momentálne

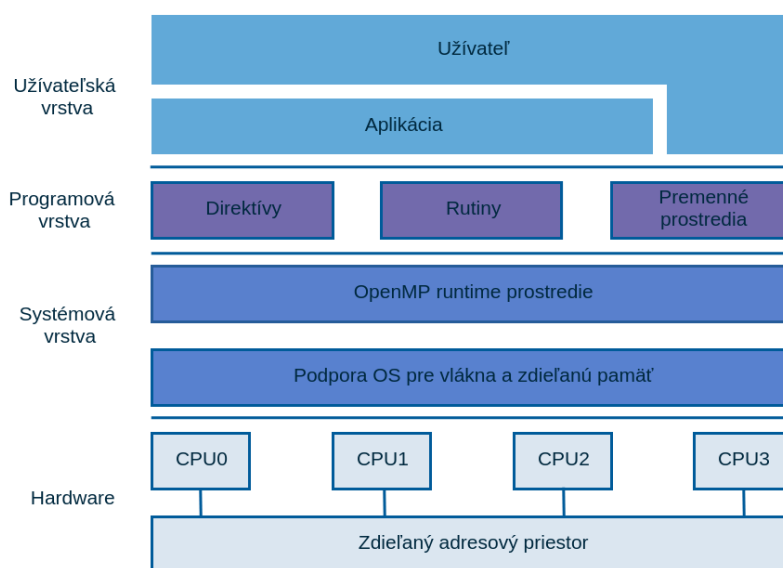
existuje API pre jazyky *C++* a *Fortran*, ktoré je dostupné pre rôzne architektúry a operačné systémy.

OpenMP podporuje jemný a hrubý paralelizmus (paralelizácia cyklov, paralelné sekcie), dátový a funkčný paralelizmus (model SPMD, paralelné sekcie). Dovoľuje inkrementálnu paralelizáciu programov so zdieľanou pamäťou, vytvárať prenositeľné a (čiastočne) škálovateľné programy. API značne zjednodušuje vytváranie viacvláknových programov, je ideálne pre viacjadrové architektúry. Od verzie *OpenMP 4.0* je doplnené o podporu akceleratorov a optimalizáciu pomocou inštrukcií SIMD.

5.2.1 Implementácia OpenMP

Vlákná *OpenMP* sú len abstrakciou nad rôznymi implementáciami vlákien. Vlákna môžu byť mapované na vlákna jadra operačného systému, vlákna *POSIX*, vlákna *Win32* apod. Program napísaný pomocou *OpenMP* je nezávislý na stroji a operačnom systéme. Pre správne fungovanie je nutné program skompilovať na cieľovej architektúre. Je dostupný pre väčšinu verzií Unix, Linux, Windows či Mac OS.

Na Obrázku 5.1 je znázornená schéma paralelného systému využívajúceho *OpenMP*. Na najnižšej úrovni je paralelný hardware pracujúci so zdieľanou pamäťou. Nad ním je vrstva operačného systému, ktorá musí podporovať vlákna a zdieľanú pamäť, spolu s "runtime" prostredím *OpenMP*. Programová vrstva zahŕňa komponenty *OpenMP* slúžiace na akceleráciu programov, ktoré budú popísané v Odseku 5.2.3. Poslednou vrstvou v hierarchii je samotná aplikácia, spolu s užívateľom, ktorý môže meniť konfiguráciu prostredia *OpenMP* pred spustením programu.



Obr. 5.1: Schéma paralelného systému využívajúceho OpenMP.

5.2.2 Princípy fungovania OpenMP

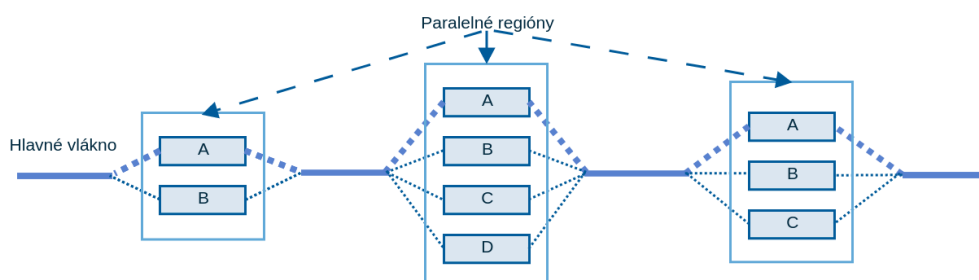
Fundamentálnym prostriedkom paralelizmu v *OpenMP* sú vlákna (paralelizmus na úrovni vlákien). Novšie verzie podporujú aj paralelizmus na úrovni inštrukcií SIMD. Vlákno je najmenšia funkčná jednotka, ktorá môže byť plánovaná operačným systémom. Vlákno je

registrované pod určitým procesom, bez ktorého nemôže existovať. Rôzne vlákna jedného procesu môžu bežať na rôznych jadrách CPU.

Typicky sa vytvára toľko pracovných vlákien, koľko je dostupných logických jadier CPU¹, API však dovoľuje nastaviť ľubovoľný počet vlákien. Vlákna sú v prípade vytvorenia počtu vlákien väčšieho ako je počet logických jadier plánované dynamicky. Výhodné je mapevať 1 vlákno na 1 logické jadro, pretože odpadá režia dynamického plánovania vlákien. Paralelizácia pomocou *OpenMP* nie je automatická, ale je plne pod kontrolou programátora (programátor explicitne určuje časti programu, ktoré budú vykonané paralelne).

Model spracovania vlákien

Časť kódu označenú na paralelné spracovanie nazývame štrukturovaný blok. Štrukturovaný blok je jeden, prípadne viac príkazov (niekoľko riadkov kódu) s jedným vstupom a jedným výstupom. Pri spracovaní programu sa využíva takzvaný "Fork – Join" model (Obrázok 5.2). Program *OpenMP* je po štarte sekvenčne vykonávaný jedným, hlavným vláknom (vlákno master). Keď dôjde k paralelnému príkazu, vytvorí sa tím vlákien (pracovné vlákna). Pracovné vlákna vykonávajú príkazy paralelne, implicitne sa synchronizujú na výstupe z paralelnej sekcie, kde sa pracovné vlákna ukončia a pokračuje len hlavné vlákno. Program sa týmto spôsobom môže vetviť ľubovoľný počet krát. *OpenMP* podporuje aj takzvaný vnořený paralelizmus, ktorý dovoľuje v rámci paralelnej skupiny vlákien vytvoriť ďalšiu skupinu vlákien.



Obr. 5.2: Model spracovania vlákien (fork – join).

5.2.3 Komponenty OpenMP

V nasledujúcej časti budú popísané hlavné komponenty *OpenMP*. Kompletná špecifikácia je dostupná v zdroji [2]. Tri hlavné súčasti *OpenMP* sú direktívy pre kompilátor, rutiny knižnice a premenné prostredia.

Direktívy kompilátora

Direktívy kompilátora² sú súčasťou zdrojového kódu, obvykle odlišené od príkazov daného programovacieho jazyka (napr. v jazyku *C++* sú to makrá začínajúce znakom #). Prekladač tieto direktívy ignoruje pokiaľ nie sú explicitne povolené (napr. prepínačom `-fopenmp`).

¹ 1 fyzické jadro môže OS vidieť ako 2 logické jadrá. Napríklad technológia Intel Hyperthreading umožňuje súčasné vykonávanie 2 vlákien (2 logické jadrá) na 1 fyzickom jadre CPU

² Direktíva – Pragma

Platnosť direktívy je obmedzená na najbližší štrukturovaný blok. Direktívy sú používané na vytvárania tímu vlákien, synchronizáciu vlákien, prípadne pre ďalšie spôsoby akcelerácie a optimalizácie kódu. Spoločný formát direktív *OpenMP* je:

`#pragma omp directive_name [clause[[,] clause]...] new_line`

Jednou z najdôležitejších je direktíva **parallel**. Keď sa program dostane na túto direktívu, z hlavného vlákna sa vytvorí tím vlákien, ktoré paralelne vykonávajú štrukturovaný blok, umiestnený pod touto direktívou. Každé vlákno má svoj identifikátor (id). Všetky vlákna vykonávajú rovnaký kód, ktorý ale môže byť podmienený pomocou id vlákna. Na konci štrukturovaného bloku nastáva implicitná synchronizácia vlákien, pričom pracovné vlákna sú ukončené a pokračuje len hlavné vlákno.

Direktíva **parallel** môže obsahovať rôzne klauzule, ktoré ovplyvňujú zdieľanie premenných, počet pracovných vlákien, afinitu vlákien³ apod. Direktíva okrem toho podporuje klauzulu **reduction(reduction-identifier : list)**, ktorá vykoná paralelnú redukciu nad definovaným zoznamom premenných s operátorom redukcie.

Veľmi užitočnou klauzulou direktívy **parallel** je **if(expression)**, kde **expression** je výraz, ktorý je vyhodnotený pred aplikovaním samotnej direktívy. Ak je výraz vyhodnotený ako pravda (nenulové číslo), tak je cyklus umiestnený pod direktívou **parallel** vykonaný paralelne. Ak je výraz vyhodnotený ako nepravda (nula), cyklus umiestnený pod direktívou **parallel** je vykonaný sekvenčne, hlavným vláknom programu.

Ďalšou veľmi dôležitou direktívou je **for**. Táto direktíva spolu s direktívou **parallel** zabezpečí rozdelenie jednotlivých iterácií cyklu **for** medzi tím vlákien. Iterácie musia byť na sebe dátovo nezávislé. Rozdelenie iterácií medzi vlákna je možné ovplyvniť pomocou klauzule **schedule(kind[, chunk_size])**. Typ plánovania môže byť *Static*, *Dynamic*, *Guided*, *Runtime*, *Auto*. Princíp jednotlivých typov plánovania je možné nájsť v dokumentácii *OpenMP* [2].

Vo Výpise 5.2 je uvedený príklad použitia rôznych direktív. Jedná sa o funkciu, ktorá vypočíta aritmetický priemer prvkov poľa pomocou paralelnej redukcie. Postupne sa vykoná nasledovné: funkcia **omp_set_num_threads** nastaví požadovaný počet vlákien, direktíva **parallel** naštartuje skupinu vlákien; direktíva **for** zabezpečí rozdelenie jednotlivých prvkov poľa medzi skupinu vlákien, pričom plánovanie je statické; ak nie je uvedené inak, všetky premenné vytvorené mimo paralelný blok sú zdieľané všetkými vláknami; premenná **i** je uvedené v klauzule **private** čo zabezpečí, že počítadlo cyklu bude pre každé vlákno privátne a jednotlivé vlákna si ho nebudú navzájom prepisovať; ako posledná je uvedená klauzula redukcie, ktorá aplikuje na prvky poľa operátor **+** a výsledok akumuluje do zdieľanej premennej **sum**.

```
1 const unsigned int NUM_THREADS = 8;
2
3 float computeAvgParallel(float *numbers, size_t elements)
4 {
5     size_t i = 0;
6     float sum = 0.0f;
7     omp_set_num_threads(NUM_THREADS);
8
9     #pragma omp parallel for private(i) schedule(static) reduction(+:sum)
10    for(i = 0; i < elements; i++)
11    {
```

³Fyzické umiestnenie vlákien na logických jadrách

```

12     sum += numbers[i];
13 }
14
15 return sum / elements;
16 }

```

Výpis 5.2: Príklad paralelného výpočtu aritmetického priemeru prvkov poľa pomocou paralelnej redukcie

Od verzie 4.0 bola pridaná podpora vektorizácie cyklov priamo pomocou *OpenMP*. Pre tento účel sa používa direktíva `simd`, ktorá optimalizuje cyklus umiestnený pod direktívou za použitia inštrukcií SIMD.

Opačný prípadom k direktíve `parallel` je `single`. Štrukturovaný blok umiestnený pod touto direktívou je vykonávaný len jedným (ľubovoľným) vláknom. Špeciálnym prípadom je direktíva `master`, kedy je kód vykonávaný jedným, špeciálne hlavným vláknom.

Iný typ paralelizmu (funkčným) získame použitím direktívy `sections`, v ktorej definujeme takzvané sekcie (`section`). Každá sekcia môže vykonávať rozdielny kód. Spolu s použitím direktívy `parallel` zabezpečíme, že jednotlivé sekcie budú bežať paralelne. Direktíva fungujúca na podobnom princípe je aj `task`, ktorá vytvorí nové vlákno pre každú úlohu.

OpenMP poskytuje prostriedky na explicitnú synchronizáciu vlákien, ktorými sú bariéra (`barrier`), kritická sekcia (`critical`), atomické operácie (`atomic`) a iné. Rôzne direktívy je možné medzi sebou kombinovať, ako bolo znázornené vo Výpise 5.2.

Rutiny OpenMP

Knižnica obsahuje okrem direktív aj rôzne rutiny určené na nastavovanie prostredia *OpenMP*, prípadne na zisťovanie aktuálneho nastavenia. Príkladmi sú `omp_get_num_threads` (získanie počtu vlákien v paralelnom regióne), `omp_set_num_threads` (nastavenie počtu vlákien v paralelnom regióne), `omp_get_thread_num` (získanie id vlákna), `omp_get_num_procs` (získanie počtu logických jadier CPU), `omp_init_lock` (inicializácia zámku), `omp_set_lock` (zamknutie zámku), `omp_unset_lock` (odmoknutie zámku), `omp_destroy_lock` (zrušenie zámku) a mnohé iné.

Premenné prostredia

Okrem direktív a podprogramov *OpenMP* je možné nastavovať rôzne vlastnosti pomocou premenných prostredia. Príklad často používaných premenných sú `OMP_NUM_THREADS` (nastavenie počtu vlákien v paralelnom regióne), `OMP_PROC_BIND` (nastavenie politiky afinity vlákien), `OMP_PLACES` (nastavenie umiestnenia vlákien), `OMP_SCHEDULE` (plánovanie paralelných cyklov), `OMP_NESTED` (povolenie/zakázanie vnoreného paralelizmu).

5.3 Knižnica HDF5

Hierarchický dátový formát *HDF5* (angl. Hierarchical Data Format) je súborový formát podporujúci vytváranie komplexných, heterogénnych súborov. Knižnica je implementovaná ako "open-source" v jazyku *C* a je často využívaná pre superpočítačové aplikácie. *HDF5* podporuje paralelný zápis/čítanie, do súboru je možné uložiť ľubovoľný dátový typ, súbory sú nezávislé na architektúre a sú samo popisné (obsahujú metadáta, ktoré popisujú štruktúru súboru).

5.4 Knižnica FFTW

*FFTW*⁴ je knižnica rutín, ktorá zabezpečuje výpočty *FFT* (angl. Fast Fourier Transform) v jednej či viac dimenziách, pre ľubovoľne veľké vstupy, pre reálne aj komplexné dáta. Knižnica je implementovaná v jazyku *C* a distribuovaná ako takzvaný voľný software. Jej rozhranie je implementované aj v matematickej knižnici *Intel MKL*⁵, ktorá transformuje volania knižnice *FFTW* na funkcie *Intel MKL* (vysoko optimalizované rutiny pre systémy Intel). Je možné využitie rôznych implementácií výpočtu *FFT* s rovnakým rozhraním, pri zmene knižnice teda nie je nutné meniť zdrojový kód.

5.5 Knižnica Google Test

Knižnica *Google Test*⁶ je *C++* framework poskytujúci prostriedky pre tvorbu jednotkových testov. Tento framework je založený na architektúre *xUnit*⁷. Knižnica je šírená ako otvorený software a je dostupná pod licenciou *BSD*. Obsahuje veľkú škálu prostriedkov pre tvorbu jednotkových testov, ktoré sú implementované prevažne ako makrá. Základnými komponentami, ktoré boli využité aj v tomto projekte sú:

1. `ASSERT_*`, `EXPECT_*` – sada makier slúžiacich na porovnávanie výsledkov testovaných jednotiek s referenčnými výsledkami. Makrá `ASSERT_*` generujú fatálnu chybu v prípade nezhody výsledkov (testovacia aplikácia sa ukončí). Makrá `EXPECT_*` v prípade nezhody výsledkov generujú iba hlásenie o chybe, pričom sa aktuálny test ukončí a pokračuje sa ďalším.
2. `TEST(TestCaseName, TestName)` – základné makro slúžiace na tvorbu testov (1 jednotkový test). Test môže byť súčasťou sady testov. Obaľuje spustenie testovaného programu a porovnanie výsledkov.
3. `TEST_F(TestCaseName, TestName)` – tvorí jeden jednotkový test (*test fixture*). Tento test patrí do takzvanej sady testov (*test suite*⁸), ktorá tvorí logický celok a zhlukuje testy podľa určitých vlastností.
4. `testing::Test` – základná trieda slúžiaca na tvorbu testovacích sád, zhlukuje testy vytvorené pomocou makier `TEST` a `TEST_F`. Testovacia sada môže definovať metódy `SetUp` a `TearDown`, ktoré sa spúšťajú pred každým, resp. po každom teste. Tieto metódy slúžia napr. na inicializáciu či deinicializáciu prostredia testu.
5. `FRIEND_TEST(TestCaseName, TestName)` – definuje konkrétny test, ako test spriatelnený s testovanou triedou (umožňuje prístup k súkromným členom triedy). Deklarácia spriatelneného testu musí byť uvedená priamo v deklarácii samotnej testovanej triedy. Využitie tohto makra je popísané v Odseku 9.1.

⁴<http://www.fftw.org/>

⁵<https://software.intel.com/en-us/mkl>

⁶<https://github.com/google/googletest>

⁷<https://en.wikipedia.org/wiki/XUnit>

⁸Názvoslovie použité vo frameworku *Google Test* je mierne odlišné od všeobecne zaužívaného. Pojem *Test* v tomto frameworku značí 1 test, ktorý sa typicky označuje ako *Test Case*. Pojem *Test Case* zase v tomto frameworku značí sadu testov, ktorá sa typicky označuje ako *Test Suite*.

6. `RUN_ALL_TESTS` – makro, ktoré spúšťa všetky registrované jednotkové testy (takzvaný *test runner*). Testy vytvorené pomocou makier `TEST` a `TEST_F` sú registrované automaticky (*test discovery*).

5.6 Kompilátory

V práci boli použité prekladače jazyka *C++* vhodné pre tvorbu vysokovýkonných aplikácií. Konkrétne sa jedná o prekladač *C++* zo sady *GCC* (GNU Compiler Collection)⁹ a prekladač *C++* vyvinutý spoločnosťou *Intel*¹⁰. Oba prekladače sú dostupné na superpočítačoch Anselm a Salomon v rôznych verziách, čo poskytuje možnosť tvorby aplikácií vo viac verziách *C++* či *OpenMP*. Počas vývoja, testovania a experimentovania s programom boli konkrétne využité kompilátory *Intel 2016*, *2017* a *GCC 5.4.0*, *6.3.0*. Požadovaný štandard *C++* je *C++11*.

5.7 Generovanie kódu za behu programu

Simulácie vo veľkých doménach, prípadne vo vysokej frekvencii si vyžadujú špeciálny prístup. Jedným z možných prístupov k urýchleniu výpočtu je generovanie kódu, ktorý bude vytvorený s ohľadom na vstupné dáta. Veľkosti domény, jej tvar, typ média, kombinácia rôznych médií, rôzne veľkosti a tvary subdomén, to všetko sú vlastnosti ovplyvňujúce náročnosť výpočtu, spotrebu pamäte apod. Simulácie s rôznymi vlastnosťami si vyžadujú rôzne prístupy, pre každú sa hodí niečo iné (iný spôsob výpočtu, iné rozdelenie dát), preto by bolo vhodné vytvorenie spustiteľného kódu, pre konkrétne vstupy.

Samozrejme nie je možné, aby vývojár vytváral špeciálny kód pre každé nové, vstupné dáta. Preto by bolo vhodné vytvorenie takzvaného pre-procesora, ktorý by na základe vstupných dát automaticky vygeneroval zdrojový, či spustiteľný kód vysoko optimalizovaný pre konkrétny prípad použitia. Samotný program by teda po spustení vygeneroval kód optimálny pre danú situáciu, ktorý by bol následne vykonaný. Aj keď takéto generovanie kódu za behu programu spotrebuje značný čas, v porovnaní s celkovou dobou výpočtu (niekoľko hodín či dokonca dní) by sa jednalo o zanedbateľné percento. Uvedený prístup ale nie je predmetom tejto práce, nakoľko je technicky veľmi náročný a v súčasnosti nie je dostupné riešenie, ktoré by vyhovovalo daným požiadavkám. Tento spôsob sa ale javí ako sľubná možnosť urýchlenia výpočtu náročných simulácií a pravdepodobne bude v blízkej dobe témou výskumu.

5.8 Dokumentácia

Celý proces vývoja 2D prototypu a finálnej, zjednotenej implementácie bol verzovaný pomocou *Git* repozitára. Vzdialený repozitár projektu bol umiestnený na súkromnom *Gitlab* serveri výskumnej skupiny *SC@FIT*. Kvôli prehľadu o aktuálnej činnosti bol priebeh implementácie dokumentovaný pomocou takzvaných "*Gitlab issues*", ktoré boli vytvorené pre každú riešenie, logickú časť programu. Jednotlivé časti implementácie (napr. prototyp 2D simulácie, implementácia jednotkových testov, finálna implementácia ...) boli verzované v samostatných vetvách *Git* repozitára. Zdrojový kód je dokumentovaný pomocou *Doxygen*.

⁹<https://gcc.gnu.org/>

¹⁰<https://software.intel.com/en-us/c-compilers>

Kapitola 6

Implementácia 2D k-Wave

Podstatným krokom pred vytvorením finálneho riešenia bolo vytvorenie návrhu a implementácie 2D simulačného programu. Nová implementácia 2D k-Wave vychádza z pôvodnej verzie 3D k-Wave, určenej pre stroje so zdieľanou pamäťou (*C++* a *OpenMP*). Je zachovaný pôvodný koncept a štruktúra kódu, popísaná v Odseku 2.2.2. Fyzikálny a matematický princíp je rovnaký ako u 3D modelu s rozdielom, že sú vynechané výpočty v dimenzii Z . Táto verzia programu je ďalej v texte označovaná ako *prototyp 2D simulácie*.

6.1 Návrh riešenia

Najzásadnejšie zmeny bolo nutné vykonať v triedach `KSpaceFirstOrder3DSolver`, `MatrixContainer`, `FftwComplexMatrix`, `OutputStreamContainer`. Z triedy `KSpaceFirstOrder3DSolver` bola vytvorená trieda `KSpaceFirstOrder2DSolver`, ktorá implementuje 2D simuláciu. Prototyp 2D simulácie je implementovaný ako samostatný program.

6.1.1 Trieda `MatrixContainer`

V tejto triede bolo nutné upraviť enumeračnú triedu `MatrixIdx` definujúcu identifikátory jednotlivých matíc. Nakoľko nová trieda pracuje len s maticami v dimenziách X a Y , bolo nutné odstrániť všetky matice pre dimenziu Z . Pôvodná trieda obsahovala napríklad identifikátory matíc hustoty v smeroch X , Y , Z , konkrétne `kRhoX`, `kRhoY`, `kRhoZ`. Nová trieda obsahuje len matice `kRhoX`, `kRhoY`. To isté platí aj pre všetky ostatné matice, ktorými sú napr. matice rýchlosti, zrýchlenia, posuv pre Fourierovu transformáciu, pomocné matice apod.

Ďalej bolo nutné upraviť metódu `addMatrices`, ktorá pridáva jednotlivé matice do kontajnera matíc. Úprava spočívala v odstránení pridávania matíc dimenzie Z do kontajnera.

6.1.2 Trieda `OutputStreamContainer`

Podobne ako v `MatrixContainer`, aj v tejto triede bolo nutné odstrániť matice, resp. výstupné prúdy dimenzie Z . Ich identifikátory boli odstránené z enumeračnej triedy `OutputStreamIdx`. Ponechané boli len prúdy dimenzií X a Y . Jednalo sa o výstupné prúdy tlaku a rýchlostí. Upravená bola aj metóda `addStreams`, z ktorej bolo odstránené pridávanie výstupných prúdov do kontajnera v dimenzii Z .

6.1.3 Trieda FftwComplexMatrix

Podstatnú časť výpočtu simulácie tvorí výpočet *FFT*. Nakoľko 2D simulácia pracuje v iných dimenziách ako pôvodná simulácia, bolo nutné upraviť spôsob výpočtu *FFT*, resp. plánovania výpočtu. Pôvodne sa počítalo 3D *FFT* v 3D priestore (pri výpočte rýchlosti a tlaku), prípadne 1D *FFT* pre výpočet posuvu rýchlosti. V novej implementácii bol nahradený výpočet 3D *FFT* za 2D *FFT*, čo znamenalo zmenu plánovania výpočtu. Boli teda vytvorené metódy plánovania `createR2CFftPlan2D`¹, `createC2RFftPlan2D`², resp. výpočtu `computeR2CFft2D`, `computeC2RFft2D`.

Ďalej bolo nutné upraviť plánovanie 1D *FFT*. U týchto metód sa nevykonáva *FFT* nad celou 3D doménou naraz, ale postupne, po 1D častiach v smere jednotlivých osí. Konkrétne sa jedná o metódy `createR2CFftPlan1DX`, `createC2RFftPlan1DX` (1D *FFT* v smere *X*) a `createR2CFftPlan1DY`, `createC2RFftPlan1DY` (1D *FFT* v smere *Y*). Plánovanie a výpočet 1D *FFT* v smere *Z* bolo odstránené.

6.1.4 Trieda KSpaceFirstOrder3DSolver

Najviac zmien si vyžadovala práve táto trieda, nakoľko sa jedná o triedu vytvárajúcu a riadiacu celú simuláciu. Keďže nová simulácia pracuje s iným dátovým modelom (dimenziionalita domény), bolo nutné upraviť všetky výpočtové jadrá a iné pomocné metódy. Celkovo bolo upravených viac ako 30 metód tejto triedy. Okrem toho boli odstránené metódy typu "getter", ktoré vracali matice dimenzie *Z* z kontajnera matíc.

Príklad 3D výpočtového jadra bol uvedený vo Výpise 2.2, Odsek 2.2.4. Rovnaká rutina upravená do verzie 2D je zobrazená vo Výpise 6.1. V kóde je možné pozorovať, že boli odstránené všetky komponenty dimenzie *Z*. Metóda nepracuje s maticami veličín v smere osi *Z*, tým pádom bol odstránený aj cyklus iterujúci cez túto dimenziu. Cyklus iterujúci cez dimenziu *Y* (vonkajší cyklus) sa stal paralelným, boli naň teda aplikované direktívy *OpenMP* (`#pragma omp parallel for schedule(static)`). Najvnútornejší cyklus je vektorizovaný pomocou direktívy `#pragma omp simd`.

```
1 // ...
2 const float* kappa = getKappa().getData();
3
4 FloatComplex* tempFftX = getTempFftwX().getComplexData();
5 FloatComplex* tempFftY = getTempFftwY().getComplexData();
6
7 FloatComplex* ddxKShiftNeg = getDdxKShiftNeg().getComplexData();
8 FloatComplex* ddyKShiftNeg = getDdyKShiftNeg().getComplexData();
9
10 #pragma omp parallel for schedule(static)
11 for (size_t y = 0; y < reducedDimensionSizes.ny; y++)
12 {
13     #pragma omp simd
14     for (size_t x = 0; x < reducedDimensionSizes.nx; x++)
15     {
16         const size_t i = get1DIndex(y, x, reducedDimensionSizes);
17         const float eKappa = divider * kappa[i];
18
19         tempFftX[i] *= ddxKShiftNeg[x] * eKappa;
20         tempFftY[i] *= ddyKShiftNeg[y] * eKappa;
```

¹R2C – Real To Complex

²C2R – Complex To Real


```

21 } // x
22 } // y
23
24 // ...

```

Výpis 6.1: Paralelná implementácia 2D výpočtového jadra

Nakoľko sú všetky viac dimenzionálne matice uložené ako lineárne (1D) pole hodnôt, tak je pre prístup k jednotlivým prvkom matice nutný výpočet indexu. Tento výpočet zabezpečuje *inline* metóda `get1DIndex`. Pôvodný výpočet v 3D bol nasledovný:

$$(z * \text{dimensionSizes.ny} + y) * \text{dimensionSizes.nx} + x$$

, kde x , y , z sú indexy do jednotlivých dimenzií a `dimensionSizes.ny` je veľkosť dimenzie Y. Keďže v 2D priestore je $z == 1$, nový výpočet je možné zjednodušiť na:

$$y * \text{dimensionSizes.nx} + x$$

, kde x , y sú indexy do jednotlivých dimenzií a `dimensionSizes.nx` je veľkosť dimenzie X.

6.2 Overenie funkčnosti a meranie výkonnosti

Vývoj 2D kódu prebiehal inkrementálne na základe pôvodného, 3D kódu. Testovanie teda prebiehalo po každej vykonanej zmene v kóde. Pôvodná implementácia využívala testovací framework *kWaveTester*, ktorý bol mierne upravený a použitý aj pre účeli testovania 2D k-Wave. Tento framework bol využitý tak isto pre testovanie finálnej implementácie a je stručne popísaný v Kapitole 9.2. Celkový koncept testovania využitý v tomto projekte je uvedený v Kapitole 9. Experimentálnemu meraniu výkonnosti prototypu 2D simulácie, ale aj finálnej implementácie sa venuje samostatná Kapitola 8.

Kapitola 7

Zjednotená implementácia 2D/3D k-Wave

Nová, zjednotená implementácia k-Wave vychádza z pôvodnej verzie 3D k-Wave, určenej pre stroje so zdieľanou pamäťou (*C++* a *OpenMP*). Jedná sa o program, ktorý rieši simuláciu ultrazvuku v 2D a 3D priestore. Simulačný program rozhoduje o type simulácie (2D alebo 3D) na základe vstupných dát. Podobne ako v Kapitole 6, aj v novej implementácii je zachovaný pôvodný koncept a štruktúra kódu, popísaná v Odseku 2.2.2. Fyzikálny a matematický princíp je rovnaký ako u 3D modelu s rozdielom, že pre 2D simuláciu sú vynechané výpočty v dimenzii Z .

7.1 Návrh riešenia

Najzásadnejšie zmeny bolo nutné vykonať v triedach `KSpaceFirstOrder3DSolver`, `MatrixContainer`, `FftwComplexMatrix`, `OutputStreamContainer`. Z triedy `KSpaceFirstOrder3DSolver` bola vytvorená trieda `KSpaceFirstOrderSolver`, ktorá je spoločná pre 2D a 3D simuláciu.

7.1.1 Trieda `MatrixContainer`

Dôležité úpravy triedy `MatrixContainer` boli vykonané najmä v metóde `void addMatrices(void)`, ktorej prototyp bol upravený do tvaru `void addMatrices(const bool is3DSolver)`. Parameter `is3DSolver` rozhoduje o tom, či sa budú do kontajnera pridávať matice uchovávajúce veličiny v dimenzii Z . Príklad inicializácie kontajnera matíc je uvedený vo Výpise 7.1. V tomto prípade nieje nutné využívať generické programovanie, nakoľko sa metóda volá iba 1 krát (počas inicializácie simulačnej triedy).

```
1 // ...
2 mContainer[MI::kRhoX].set(MT::kReal, fullDims, kNoLoad, kCheckpoint, kRhoXName);
3 mContainer[MI::kRhoY].set(MT::kReal, fullDims, kNoLoad, kCheckpoint, kRhoYName);
4 if (is3DSolver)
5 {
6     mContainer[MI::kRhoZ].set(MT::kReal, fullDims, kNoLoad, kCheckpoint, kRhoZName);
7 }
8 // ...
```

Výpis 7.1: Príklad inicializácie kontajnera matíc, metóda `void addMatrices(const bool is3DSolver)`

7.1.2 Trieda OutputStreamContainer

Podobne ako v predchádzajúcom prípade, aj v triede `OutputStreamContainer` bolo nutné upraviť metódu `void addMatrices(void)`, ktorá inicializuje kontajner výstupných prúdov (matic). Princíp úpravy bol rovnaký ako v triede `MatrixContainer`, kde bol pridaný parameter rozhodujúci o dimenzionalite simulácie.

7.1.3 Trieda FftwComplexMatrix

Pôvodná trieda zabezpečujúca výpočet *FFT* pracovala s 3D maticami. V prvom prototype 2D simulácie bola táto trieda upravená pre 2D model (Odsek 6.1.3). V zjednotenom riešení je nutné, aby trieda poskytovala rozhranie pre výpočet *FFT* nad 2D a 3D dátami. V triede bolo implementovaných asi 10 nových metód, ktoré obaľujú volania funkcií z knižnice *FFTW*. Nachádzajú sa tu teda metódy pre plánovanie a výpočet *FFT* nad celými 2D a 3D maticami, ďalej nad 2D maticami v smeroch osi *X* a *Y*, nad 3D maticami v smeroch osi *X*, *Y* a *Z*. Metódy implementujú výpočet *FFT* v oboch smeroch (prevod reálnych čísel na spektrum a prevod spektra na reálne čísla).

7.1.4 Trieda KSpaceFirstOrderSolver

Táto trieda je jadrom celého simulačného programu. Pre súčasnú podporu 2D a 3D simulácie bolo nutné vykonať veľké množstvo zmien. Kód všetkých výpočtových jadier musel byť upravený tak, aby podporoval výpočet nad 2D a 3D dátovým modelom, pričom bolo nutné zachovať štruktúru, prehľadnosť a efektivitu kódu. Zásadným krokom pre dodržanie týchto podmienok bolo využitie generického programovania a vytvorenie takzvanej šablóny triedy `KSpaceFirstOrderSolver`. Problematika generického programovania bola popísaná v Odseku 5.1.2. Vo Výpise 7.2 je naznačená deklarácia triedy `KSpaceFirstOrderSolver`, ktorá je parametrizovaná parametrom šablóny `TIs3d` typu `bool`. Tento parameter určuje, či sa jedná o 3D simuláciu (`TIs3d == true`) alebo 2D simuláciu (`TIs3d == false`). Predvolená hodnota parametru je `true`.

```
1 template<bool TIs3d = true>
2 class KSpaceFirstOrderSolver: public KSpaceFirstOrder
3 {
4 public:
5     /// Constructor.
6     KSpaceFirstOrderSolver();
7
8     // ...
9 };
```

Výpis 7.2: Šablóna triedy `KSpaceFirstOrderSolver`

Vo výpise 7.2 je vidieť, že trieda `KSpaceFirstOrderSolver` dedí z básovej triedy `KSpaceFirstOrderSolverBase`, ktorá definuje rozhranie. Vytvorenie básovej triedy je dôležité z hľadiska unifikovaného prístupu k verejným metódam triedy `KSpaceFirstOrderSolver` pre 2D a 3D simuláciu. Základná trieda je definovaná v hlavičkovom súbore `KSpaceFirstOrderSolverBase.h`. Jej význam bude bližšie popísaný v závere tejto podkapitoly.

Výpočtové jadrá využívajú práve parameter `TIs3d` na určenie dimenzionality simulácie. Jednoduché výpočtové jadro, ktoré slúži na výpočet počiatočnej akustickej rýchlosti je

uvedené vo výpise 7.3. Parameter `TIs3d` je predávaný automaticky do každej metódy triedy v dobe kompilácie programu. Podmienené príkazy závislé na danom parametre sú teda vyhodnotené počas prekladu programu, čo je v našom prípade dôležité pre dosiahnutie maximálneho výkonu. Kód na 20. riadku využíva parameter `TIs3d` pri získavaní referencie na maticu `dzudznSgz`. Ak sa jedná o 3D doménu, nastaví sa správna referencia na maticu `dzudznSgz`. V opačnom prípade sa nastaví referencia na maticu `dxudxnSgx`, ktorá sa ďalej v kóde ignoruje, pretože sa pri výpočte nepoužíva.

Dôležitá časť kódu je uvedená na 27. riadku, kde sa spúšťa paralelný cyklus nad dimenziou Z len v prípade, ak sa jedná o 3D doménu. V opačnom prípade sa paralelný cyklus vykonáva nad dimenziou Y (riadok 31). Pre tento účel je použitá klauzula `if` direktívy `omp parallel`, ktorej význam je popísaný v Kapitole 5.2.3. Na riadku 42 je uvedený výpočet novej hodnoty `uzSgz` v prípade, že sa jedná o 3D doménu. Vďaka tomu, že hodnota `TIs3d` je známa v dobe prekladu programu, môže byť najvnútornejší cyklus bez problémov vektorizovaný pomocou direktívy `omp simd`.

```

1 template<bool TIs3d>
2 void KSpaceFirstOrderSolver<TIs3d>::computeInitialVelocityHomogeneousNonuniform()
3 {
4     getTempFftwX().computeC2RfftND(getUxSgx());
5     getTempFftwY().computeC2RfftND(getUySgy());
6     // výpočet FFT v dimenzii Z v prípade 3D simulácie
7     if (TIs3d)
8         getTempFftwZ().computeC2RfftND(getUzSgz());
9
10    const DimensionSizes& dimensionSizes = mParameters.getFullDimensionSizes();
11    const size_t nElements = dimensionSizes.nElements();
12
13    const float dividerX = 1.0f / (2.0f * static_cast<float>(nElements)) * mParameters.
        getDtRho0SgxScalar();
14    const float dividerY = 1.0f / (2.0f * static_cast<float>(nElements)) * mParameters.
        getDtRho0SgyScalar();
15    const float dividerZ = 1.0f / (2.0f * static_cast<float>(nElements)) * mParameters.
        getDtRho0SgzScalar();
16
17    const RealMatrix& dxudxnSgx = getDxudxnSgx();
18    const RealMatrix& dyudynSgy = getDyudynSgy();
19    // využitie ternárneho operátora pre získanie referencie na maticu dzudznSgz
20    const RealMatrix& dzudznSgz = (TIs3d ? getDzudznSgz() : getDxudxnSgx());
21
22    RealMatrix& uxSgx = getUxSgx();
23    RealMatrix& uySgy = getUySgy();
24    RealMatrix& uzSgz = (TIs3d ? getUzSgz() : getUxSgx());
25
26    // paralelný cyklus cez dimenziu Z, ak sa jedná o 3D simuláciu
27    #pragma omp parallel for schedule(static) if(TIs3d)
28    for (size_t z = 0; z < dimensionSizes.nz; z++)
29    {
30        // paralelný cyklus cez dimenziu Y, ak sa jedná o 2D simuláciu
31        #pragma omp parallel for schedule(static) if(!TIs3d)
32        for (size_t y = 0; y < dimensionSizes.ny; y++)
33        {
34            #pragma omp simd
35            for (size_t x = 0; x < dimensionSizes.nx; x++)
36            {
37                const size_t i = get1DIndex(z, y, x, dimensionSizes);
38                uxSgx[i] *= dividerX * dxudxnSgx[x];

```

```

39     uySgy[i] *= dividerY * dyudynSgy[y];
40     // výpočet v dimenzii Z, len ak sa jedná o 3D simuláciu
41     if (TIs3d)
42         uzSgz[i] *= dividerZ * dzudznSgz[z];
43     } // x
44 } // y
45 } // z
46 }

```

Výpis 7.3: Príklad výpočtového jadra šablónovej triedy KSpaceFirstOrderSolver

Uvedený kód je príkladom jednoduchého výpočtového jadra, ktoré je chránenou metódou šablónovej triedy KSpaceFirstOrderSolver. Volanie metódy `computeInitialVelocityHomogeneousNonuniform` v rámci triedy je zobrazené vo Výpise 7.4. Nakoľko je parameter `TIs3d` súčasťou inštancie samotnej triedy, nieje nutné ho explicitne predávať pri volaní jednotlivých metód šablónovej triedy.

```

1 // ...
2 if (mParameters.getNonUniformGridFlag())
3 {
4     // non uniform grid, homogeneous case
5     computeInitialVelocityHomogeneousNonuniform();
6 }
7 // ...

```

Výpis 7.4: Príklad volania výpočtového jadra v rámci šablónovej triedy KSpaceFirstOrderSolver

Po úprave prototypu triedy KSpaceFirstOrderSolver bola taktiež nutná úprava spôsobu vytvárania jej objektov. Spôsob inštanciacie objektov danej triedy je uvedený vo Výpise 7.5. Po spustení programu sú ako prvé spracované argumenty zadané do príkazového riadku. Z argumentov je získaný názov súboru obsahujúci vstupné dáta. Zo súboru sú ešte pred vytvorením inštancie simulačnej triedy načítané skalárne hodnoty, ktoré ovplyvňujú chovanie simulácie. V tomto bode sú dôležité rozmery vstupnej domény, z ktorých sa zisťuje, či sa jedná o 2D alebo 3D simuláciu. Parametre simulácie sú spracované a uložené pomocou triedy `Parameters`, ktorá je popísaná v Odseku 2.2.2.

Na základe dimenzionality domény je následne vytvorená inštancia simulačnej triedy. V prípade 3D simulácie je konštruktoru šablónovej triedy predaná hodnota `true`, v opačnom prípade `false`. Inštancia triedy KSpaceFirstOrderSolver<true> je pre kompilátor objekt odlišný od objektu KSpaceFirstOrderSolver<false>. Nieje teda možné priradiť objekt typu KSpaceFirstOrderSolver<true> do premennej typu KSpaceFirstOrderSolver<false> a naopak. Pre tento účel je vhodné využiť polymorfizmus, pričom je možné ukazovateľ na oba typy priradiť do premennej typu KSpaceFirstOrderSolverBase* (ukazovateľ na базовú triedu, riadok 17 a 21). Kvôli automatickej správe pamäte je využitý koncept z knižnice *STL* – `std::shared_ptr<T>` ("inteligentný" ukazovateľ). Veľmi dôležitým faktom je, že kompilátor musí vytvoriť oba objekty šablónovej simulačnej triedy (KSpaceFirstOrderSolver<true> a KSpaceFirstOrderSolver<false>), teda vygenerovať kód pre 2D a 3D simuláciu. Vyhodnocovanie podmienok závislých ma parametre `TIs3d` vo výpočtových jadrách sa teda za behu programu vôbec nevykonáva, čo umožňuje kompilátoru optimalizáciu kódu.

```

1 // ...
2 // spracovanie argumentov programu; načítanie skalárnych hodnôt zo vstupného súboru
3 try
4 {
5     parameters.init(argc,argv);
6 }
7 catch (const std::exception &e)
8 {
9     // ošetrenie výnimiek
10 }
11 // Vytvorenie inštancie triedy KSpaceFirstOrderSolver na základe dimenzionality domény
12 std::shared_ptr<KSpaceFirstOrderSolverBase> kSpaceSolver;
13 try
14 {
15     if(parameters.getFullDimensionSizes().is2D())
16     {
17         kSpaceSolver = std::shared_ptr<KSpaceFirstOrderSolverBase>(new KSpaceFirstOrderSolver<
18             false>());
19     }
20     else
21     {
22         kSpaceSolver = std::shared_ptr<KSpaceFirstOrderSolverBase>(new KSpaceFirstOrderSolver<
23             true>());
24     }
25 }
26 catch(const std::bad_alloc& e)
27 {
28     // ošetrenie výnimiek
29 }
30 // volanie jednotlivých metód rozhrania KSpaceFirstOrderSolverBase
31 try
32 {
33     kSpaceSolver->allocateMemory();
34 }
35 catch (const std::bad_alloc& e)
36 {
37     // ošetrenie výnimiek
38 }
39 // ...

```

Výpis 7.5: Inštanciácia šablónovej triedy KSpaceFirstOrderSolver

7.1.5 Vylepšenia triedy KSpaceFirstOrderSolver

Súčasťou práce bola aj optimalizácia kódu z hľadiska jeho výkonu, štruktúry či prehľadnosti. Od prvej verzie k-Wave prešli prekladače jazyka *C++* vývojom, ktorý priniesol množstvo nových vlastností. Dnešné prekladače dovoľujú optimalizáciu zložitejších konštrukcií *C++*, ako tomu bolo počas prvého vývoja projektu k-Wave.

Prístup k dátam maticových tried

Väčšina dát s ktorými pracuje simulačný program k-Wave je uložená v podobe matíc. Matice sú ďalej organizované v takzvanom kontajnere matíc (Odsek 2.2.2). Prístup k jednotlivým prvkom matice môže byť realizovaný viacerými spôsobmi. Vo Výpise 7.6 je uvedený príklad konštrukcie získavania dát z maticových objektov použitý v celom projekte. Na 2. riadku

je znázornený pôvodný koncept, kde sa k dátam matice prístupuje cez "surový" ukazovateľ na dáta typu `float` pomocou metódy `getData`. Tento prístup bol nutný kvôli využitiu inštrukcií typu `SIMD`, teda vektorovému spracovaniu dát. Staršie kompilátory nepodporovali vektorizáciu kódu pracujúceho nad dátami zapúzdrenými v rámci `C++` triedy. Tento koncept bol optimálny z hľadiska výkonu, ale porušoval princípy objektovo orientovaného programovania súvisiace so zapúzdrením dát.

Nový koncept využíva pokročilé vlastnosti kompilátorov, ktoré sú schopné vektorizovať aj zložitejšie konštrukcie. Na riadku 11 je znázornené získanie referencie na maticu z kontajnera. K samotným dátam matice sa následne neprístupuje cez "surový" ukazovateľ, ale cez operátor `[]`. Operátor pre prístup k dátam matice je definovaný v rámci tried `RealMatrix`, `ComplexMatrix`, `IndexMatrix`. Operátor vracia referenciu na prvky matice na základe ich indexu. Tento koncept je bližší princípom OOP a je podporovaný dnešnými kompilátormi (v rámci diplomovej práce boli využívané napr. prekladače *Intel 2017*, *GCC 6.3.0*, prípadne novšie). Kľúčovou podmienkou pre využitie nového konceptu prístupu k dátam bolo zachovanie minimálne pôvodného výkonu programu. Rýchlosť výpočtu je do veľkej miery ovplyvnená úspešnosťou vektorizácie kritických `for` cyklov. Je dôležité uviesť, že operátor musí byť implementovaný v hlavičkovom súbore ako *inline* metóda.

```

1  const size_t nElements = mParameters.getFullDimensionSizes().nElements();
2  // získanie ukazovateľa na dáta matice
3  float *uxSgxPtr = getUxSgx().getData();
4  // prístup k dátam pomocou ukazovateľa
5  #pragma omp parallel for schedule(static)
6  for (size_t i = 0; i < nElements; i++)
7  {
8      uxSgxPtr[i] = 0.0f;
9  }
10
11 // získanie referencie objektu matice
12 RealMatrix& uxSgxRef = getUxSgx();
13 // prístup k dátam pomocou prístupového operátora definovaného v triede RealMatrix
14 #pragma omp parallel for schedule(static)
15 for (size_t i = 0; i < nElements; i++)
16 {
17     uxSgxRef[i] = 0.0f;
18 }

```

Výpis 7.6: Dva rôzne spôsoby prístupu k dátam maticových tried

Kompilátor môže počas prekladu kódu generovať správu o vykonaných optimalizáciách. Napr. pre kompilátor *Intel* je možné túto možnosť povoliť pomocou prepínača `-qopt-report`. Týmto spôsobom je možné exaktne overiť, ktoré optimalizácie boli vykonané na jednotlivých častiach kódu. Tento postup overenia bol aplikovaný na všetky výpočtové jadrá triedy `KSpaceFirstOrderSolver`. Ako ukážka overenia vektorizácie je vo výpise 7.7 uvedená časť optimalizačnej správy kompilátora *Intel*, ktorá sa zaoberá metódou

`KSpaceFirstOrderSolver::computeVelocityGradient`. Na riadku 9 a 10 je uvedený oznam o vektorizácii najvnútornejšieho cyklu spomínanej metódy (odkazuje sa na konkrétny riadok v zdrojovom súbore `KSpaceFirstOrderSolver.cpp`, v tomto prípade riadok 1022). Zachovanie minimálne pôvodného výkonu bolo taktiež overené experimentálne, pričom jednotlivé verzie kódu boli porovnávané na simuláciách rôznych veľkostí (Odsek 8.1, obrázky 8.1 a 8.2).

```

1 LOOP BEGIN at KSpaceSolver/KSpaceFirstOrderSolver.cpp(1019,5)
2   remark #15542: loop was not vectorized: inner loop was already vectorized
3
4   LOOP BEGIN at KSpaceSolver/KSpaceFirstOrderSolver.cpp(1022,7)
5   <Peeled loop for vectorization>
6     remark #25456: Number of Array Refs Scalar Replaced In Loop: 3
7   LOOP END
8
9   LOOP BEGIN at KSpaceSolver/KSpaceFirstOrderSolver.cpp(1022,7)
10    remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
11  LOOP END
12
13  LOOP BEGIN at KSpaceSolver/KSpaceFirstOrderSolver.cpp(1022,7)
14  <Remainder loop for vectorization>
15    remark #15301: REMAINDER LOOP WAS VECTORIZED
16  LOOP END
17
18  LOOP BEGIN at KSpaceSolver/KSpaceFirstOrderSolver.cpp(1022,7)
19  <Remainder loop for vectorization>
20    remark #25456: Number of Array Refs Scalar Replaced In Loop: 3
21  LOOP END
22 LOOP END

```

Výpis 7.7: Časť optimalizačného výpisu kompilátora Intel, overenie vektorizácie najvnútornejšieho cyklu metódy `computeVelocityGradient`

Volanie šablónových metód

V projekte `k-Wave` je využitý generické programovanie aj na úrovni šablón funkcií (metód). Základný princíp tohto konceptu bol popísaný v Odseku 5.1.2. Skrátený príklad šablónovej metódy implementovanej v triede `KSpaceFirstOrderSolver` je uvedený vo Výpise 7.8. Uvedená metóda je parametrizovaná pomocou 2 parametrov šablóny typu (`bool`). Tieto parametre rozhodujú o tom, či sa pri výpočtoch použijú skalárne hodnoty daných veličín alebo matice. Na konci uvedeného výpisu je znázornený spôsob volania metódy. Tento spôsob implementácie bol zvolený kvôli nutnosti dosiahnutia maximálneho výkonu (parametre sú známe počas prekladu programu), ale nie je ideálny z hľadiska prehľadnosti kódu. V projekte sa okrem toho nachádzajú aj metódy parametrizované až 3 parametrami, čo znamená že môže nastať až 8 prípadov volania metódy. Volania takýchto metód sa navyše v kóde opakujú, čo vedie na vysokú redundanciu kódu.

```

1 template<bool TIs3d>
2 template<bool c0ScalarFlag, bool areTauAndEtaScalars>
3 void KSpaceFirstOrderSolver<TIs3d>::sumPressureTermsNonlinear()
4 {
5   // ...
6   // získanie referencie na maticu alebo skalárnej hodnoty
7   // podľa parametrov c0ScalarFlag a areTauAndEtaScalars
8   const float c2Scalar = (c0ScalarFlag) ? mParameters.getC2Scalar() : 0;
9   const RealMatrix& c2Matrix = (c0ScalarFlag) ? getTemp2Real3D() : getC2();
10  const float absorbTauScalar = (areTauAndEtaScalars) ? mParameters.getAbsorbTauScalar() :
11    0;
12  const RealMatrix& absorbTauMatrix = (areTauAndEtaScalars) ? getTemp2Real3D() :
    getAbsorbTau();

```



```

13 #pragma omp parallel for simd schedule(static)
14 for (size_t i = 0; i < nElements; i++)
15 {
16     // použitie skalárnej hodnoty, alebo prvkov matice
17     const float c2 = (c0ScalarFlag) ? c2Scalar : c2Matrix[i];
18     const float absorbTau = (areTauAndEtaScalars) ? absorbTauScalar : absorbTauMatrix[i];
19     // ...
20 }
21 }
22
23 // príklad volania metódy
24 if (mParameters.getC0ScalarFlag())
25 {
26     if (mParameters.getAlphaCoeffScalarFlag())
27     {
28         sumPressureTermsNonlinear<true, true>();
29     }
30     else
31     {
32         sumPressureTermsNonlinear<true, false>();
33     }
34 }
35 else
36 {
37     sumPressureTermsNonlinear<false, false>();
38 }

```

Výpis 7.8: Metóda parametrizovaná pomocou parametrov šablóny

Kvôli zvýšeniu prehľadnosti kódu bol zavedený nasledovný spôsob volania metód. Bola implementovaná metóda `initializeMethodPointers`, ktorá inicializuje asociatívnu mapu uchovávajúcu odkazy na jednotlivé metódy. Kľúčom do mapy je dvojica (prípadne trojica) hodnôt typu `bool` a hodnotou je ukazovateľ na jednotlivé metódy. Spôsob vytvorenia asociatívnej mapy a príklad volania metódy je naznačený vo Výpise 7.9. Na 6. riadku je definovaný typ ukazovateľ na metódu `sumPressureTermsNonlinear`, nasledované naplnením asociatívnej mapy správnymi hodnotami. Na riadku 11 je uvedené získanie ukazovateľa na príslušnú metódu podľa parametrov `c0ScalarFlag` a `alphaCoeffScalarFlag` (nahradzuje vetvenie programu vo Výpise 7.8 riadok 24) nasledované volaním metódy.

```

1 template<bool TIs3d>
2 void KSpaceFirstOrderSolver<TIs3d>::initializeMethodPointers()
3 {
4     using s = KSpaceFirstOrderSolver;
5     // definovanie typu ukazovateľ na metódu
6     typedef void (s::*SumPressTermsNonlinPtr)();
7     // vytvorenie mapy
8     std::map<std::tuple<bool, bool>, SumPressTermsNonlinPtr> methodPtr;
9     methodsPtr[make_tuple(true, true)] = &s::sumPressureTermsNonlinear<true,true>;
10    methodsPtr[make_tuple(true, false)] = &s::sumPressureTermsNonlinear<true,false>;
11    methodsPtr[make_tuple(false, true)] = &s::sumPressureTermsNonlinear<false,true>;
12    methodsPtr[make_tuple(false, false)] = &s::sumPressureTermsNonlinear<false,false>;
13    // ...
14    // vytvorenie asociatívnej mapy pre každú šablónovú metódu
15    // ...
16 }
17

```



```

18 // príklad získania ukazovateľa na metódu a jej volanie
19 SumPressTermsNonlinPtr sumPtr = methodsPtr[std::make_tuple(mParameters.getC0ScalarFlag(),
    mParameters.getAlphaCoeffScalarFlag())];
20 (this->*sumPtr)();

```

Výpis 7.9: Asociatívna mapa uchováajúca odkazi na šablónové metódy

7.2 Overenie funkčnosti a meranie výkonnosti

Podobne ako pri implementácii 2D prototypu, aj v tomto prípade bol pre overenie funkčnosti použitý testovací framework *kWaveTester* 9.2. Okrem toho bol navrhnutý a implementovaný spôsob jednotkového testovania, ktorý je popísaný v Odseku 9.1. Experimentálne meranie výkonnosti finálnej implementácie (spolu s prototypom 2D simulácie) je popísané v Kapitole 8.

Kapitola 8

Experimentálne výsledky

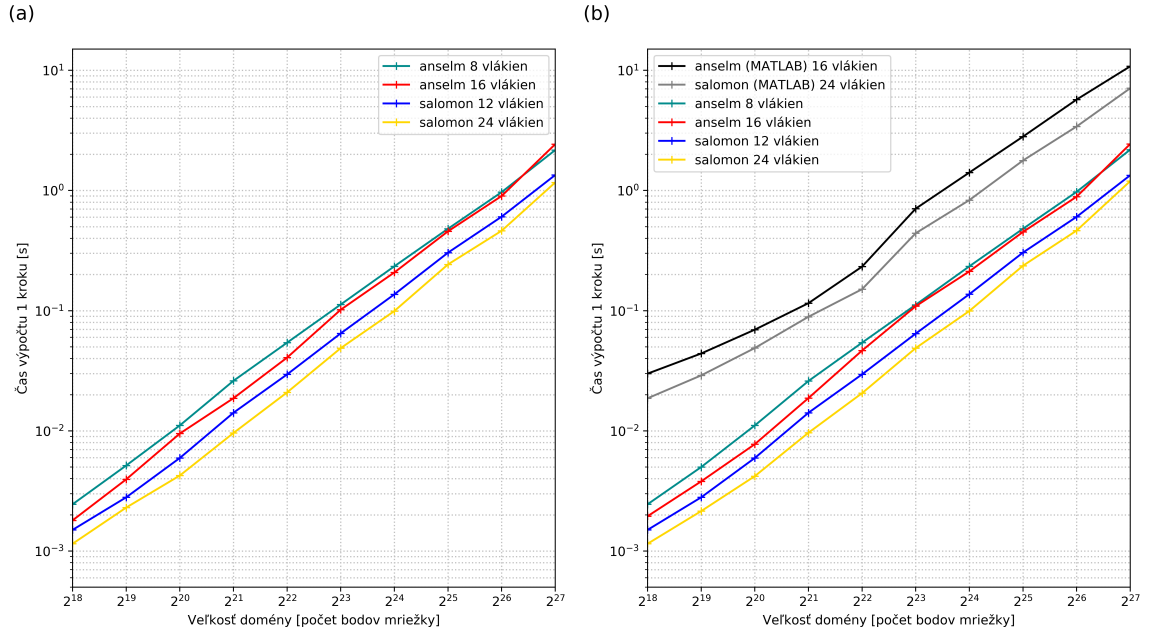
Meranie výkonnosti prebiehalo experimentálne na superpočítačoch Anselm a Salomon. Testovanie bolo zamerané na škálovanie nových programov a porovnanie výkonu jednotlivých implementácií s rôznymi testovacími vstupmi. V tejto kapitole sa nachádzajú výsledky pre oba programy, ktoré vznikli v rámci diplomovej práce (prototyp 2D implementácie a výsledná, zjednotená 2D/3D implementácia). Ku každému grafu bude uvedené, o aký program sa jedná a na akom prostredí bol spustený. Výsledky každého experimentu boli merané opakovane (minimálne 3 krát) z dôvodu získania presnejších výsledkov. Keďže sa výsledky jednotlivých meraní líšili len minimálne, výsledná hodnota (napr. doba výpočtu simulácie) bola vypočítaná ako priemer nameraných hodnôt. Výsledné hodnoty boli následne použité pre tvorbu nižšie uvedených grafov.

Pre tvorbu vstupov simulácie (vstupného *HDF5* súboru) bol využitý testovací framework *kWaveTester* (Odsek 9.2). Generované médium vstupnej domény bolo heterogénne a stratové (absorpčné). Šírenie akustickej vlny bolo zvolené ako nelineárne.

8.1 Doba výpočtu jedného kroku simulácie

Na Obrázku 8.1 sú grafy znázorňujúce čas výpočtu jedného kroku simulácie pre rôzne veľkosti 2D domény. Obrázok 8.1a znázorňuje výsledky pre prvý prototyp 2D implementácie, Obrázok 8.1b zase výsledky pre finálnu, zjednotenú implementáciu. Veľkosti domény sa pohybujú od 2^{18} (512x512) do 2^{27} (8192x16384) bodov mriežky. Veľkosť vstupného súboru pre doménu 2^{18} je 9 MB. Pre najväčšiu testovanú doménu (2^{27}) je to 4 GB. Nakoľko sú oba obrázky (teda aj časy výpočtu) veľmi podobné, bude popísaný graf (b), teda finálna implementácia. Cieľom tohto testu bolo zistiť, či sa výrazné zmeny v kóde zjednotenej 2D/3D simulácie (popísané v Kapitole 7) negatívne prejavajú na výkone. Zistilo sa, že použité princípy (najmä využitie generického programovania) nemajú negatívny vplyv na výkon programu.

Časy výpočtu jedného kroku simulácie sa pre najmenšiu doménu pohybujú približne v intervale 0.001 s (Salomon 24 vlákien) až 0.004 s (Anselm 8 vlákien). Nakoľko sú na grafe obe osi v logaritmickej merítke, je možné vidieť 4 lineárne priamky. Čas výpočtu jedného kroku sa približne zdvojnásobí pri zdvojnásobení veľkosti domény. Rozdieli času výpočtu medzi superpočítačmi Salomon a Anselm rozhodne nie sú zanedbateľné. Ak si zoberieme napríklad doménu o veľkosti 2^{27} , výpočet jedného kroku na jednom uzly Salomon (2xCPU, 24 vlákien) trvá približne 1.2 s. Na jednom uzle superpočítača Anselm trvá rovnaký výpočet 2.5 s, čo je viac ako dvojnásobok. Simulácia, ktorej výpočet by na Salomone trval napr. 24 hodín,

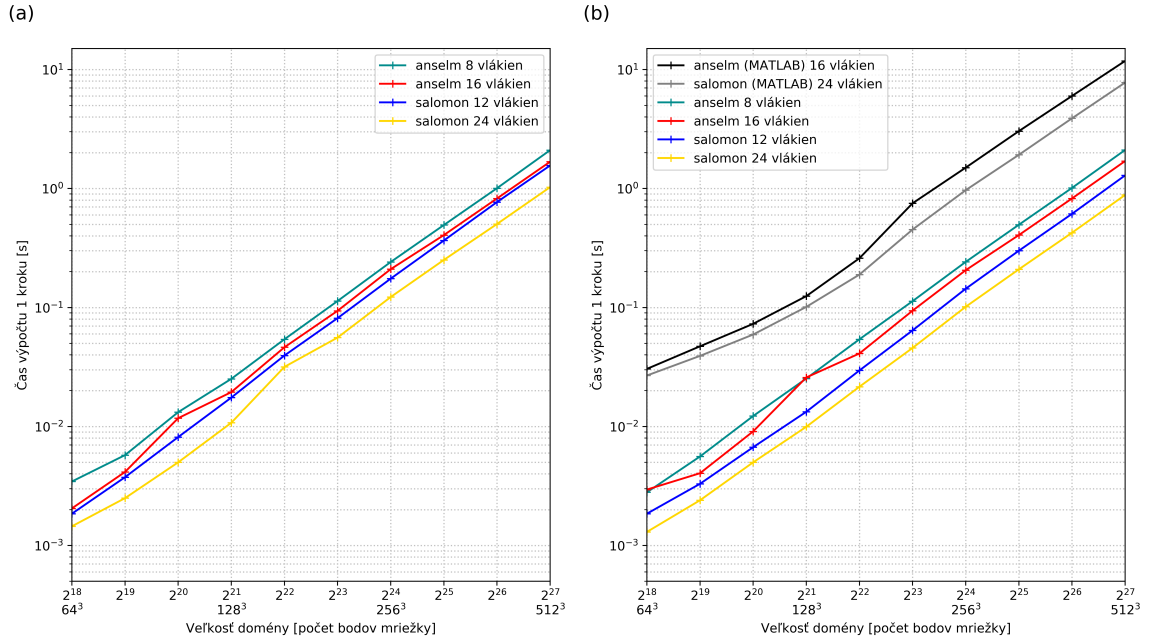


Obr. 8.1: Čas výpočtu jedného kroku simulácie pre rôzne veľkosti 2D domény, (a) prototyp 2D implementácie, (b) finálna 2D/3D implementácia, superpočítače Anselm a Salomon. V grafe sú osi X a Y v logaritmickom merítku.

by na Anselme trval asi 50 hodín. Hlavnou príčinou tohto rozdielu sú odlišné procesory. Procesory jedného uzla Salomonu majú o 50 % viac jadier ako uzol Anselmu a základná frekvencia je vyššia o 100 MHz (Turbo boost o 200 MHz). Procesory na Salomone využívajú navyše inštrukčnú sadu AVX2, pričom na Anselme je to staršia technológia – AVX.

Dôležitou úlohou je overenie zrýchlenia voči *MATLAB* implementácii 2D k-Wave, ktorej časy výpočtu jedného kroku simulácie sú znázornené taktiež na Obrázku 8.1b. *MATLAB* k-Wave využíva celý uzol superpočítača, teda 16 vlákien pre Anselm a 24 pre Salomon. Pri porovnaní časov výpočtu kroku *C++* a *MATLAB* simulácie napr. nad doménou 4096x4096, dosahuje prvá z uvedených čas približne 0.1 s a druhá z uvedených 0.83 s, čo znamená viac ako osemnásobné zrýchlenie (Salomon). Pre rovnakú veľkosť vstupu a superpočítač Anselm je zrýchlenie približne sedemnásobné.

Na Obrázku 8.2 sú znázornené časy výpočtu kroku simulácie pre rôzne veľkosti 3D domény (rovnaké ako v predchádzajúcom prípade). Cieľom tohto experimentu bolo overiť a porovnať výkon pôvodnej 3D implementácie (Obrázok 8.2a) a novej zjednotenej implementácie (Obrázok 8.2b). Zistilo sa, že výkon novej implementácie nieje horší ako výkon pôvodnej, čo bolo jednou z najdôležitejších podmienok. Naopak je možné pozorovať, že napr. pre doménu o veľkosti 2^{27} trval výpočet jedného kroku simulácie na superpočítači Salomon (24 vlákien) 1.025 s pre pôvodnú 3D implementáciu a 0.91 s pre novú zjednotenú implementáciu. Nedá sa hovoriť o výraznom zrýchlení výpočtu, pretože musíme brať do úvahy aj nepresnosť merania. Každopádne z experimentu vyplýva, že rýchlosť výpočtu novej, zjednotenej implementácie nieje horšia ako u pôvodného programu. Na Obrázku 8.2b sú opäť uvedené aj časy výpočtu simulácie pomocou 3D *MATLAB* k-Wave. Dosiahnuté zrýchlenie novej *C++* implementácie voči *MATLAB* k-Wave je pre najväčšiu doménu približne 7 na superpočítači Anselm a približne 8.5 na superpočítači Salomon.



Obr. 8.2: Čas výpočtu jedného kroku simulácie pre rôzne veľkosti 3D domény, (a) pôvodná 3D implementácia, (b) finálna 2D/3D implementácia, superpočítače Anselm a Salomon. V grafe sú osi X a Y v logaritmickom merítke.

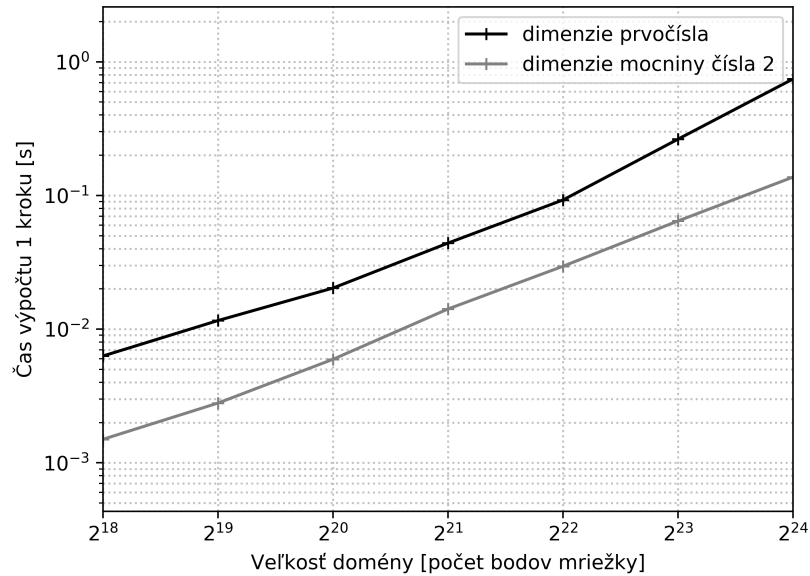
8.1.1 Vplyv rozmerov domény na čas výpočtu

Väčšina vykonaných experimentálnych meraní pracuje s doménami, ktorých dimenzie majú rozmery mocniny čísla 2. Takéto rozmery sú všeobecne vhodné z hľadiska zarovnania dát v pamäti, vektorizácie či typu použitého algoritmu. Na Obrázku 8.3 je uvedené porovnanie výkonu programu, ktorý je závislý práve na rozmeroch vstupných dát. Jedná sa o jednu verziu programu (finálna implementácia), ktorá bola spustená nad rôznymi dátami. Sivá krivka zobrazuje čas výpočtu pre domény, ktorých veľkosti dimenzií sú mocniny čísla 2 (512×512 , 512×1024 , 1024×1024 ...). Čierna krivka zobrazuje zase čas výpočtu pre domény, ktorých veľkosti dimenzií sú prvočísla blízke uvedeným mocninám čísla 2 (521×521 , 521×1021 , 1021×1021 , 1021×2039 , 2039×2039 , 2039×4099 , 4099×4099).

Z nameraných výsledkov vychádza, že výpočet nad doménou s prvočíselnými rozmermi trvá 3 až 5.5 krát viac (platí len pre uvedené vstupy, pre iné rozmery sa výsledky môžu líšiť), ako nad vstupom s rozmermi mocniny čísla 2. Tento fakt je spôsobený najmä tým, že veľkú časť celkového času tvorí výpočet *FFT* (popísané v Odseku 8.4), ktorý je závislý na rozmeroch vstupných dát (pre výpočet nad vstupmi s prvočíselnými rozmermi sa používa algoritmus, ktorý má vyššiu časovú zložitosť, závisí to od konkrétnej implementácie). Pre maximálnu efektivitu výpočtu je teda potrebné (pokiaľ je to možné) voliť vstupy s vhodnými rozmermi.

8.2 Škálovanie výpočtu

Veľmi zaujímavou a dôležitou metrikou je škálovanie výpočtu na väčšom počte vlákien, resp. zrýchlenie výpočtu voči 1 vláknu. Nasledujúce experimenty pracujú s finálnou, zjednotenou implementáciou 2D/3D simulácie. Ako prvé je uvedené škálovanie na superpočítači Anselm,

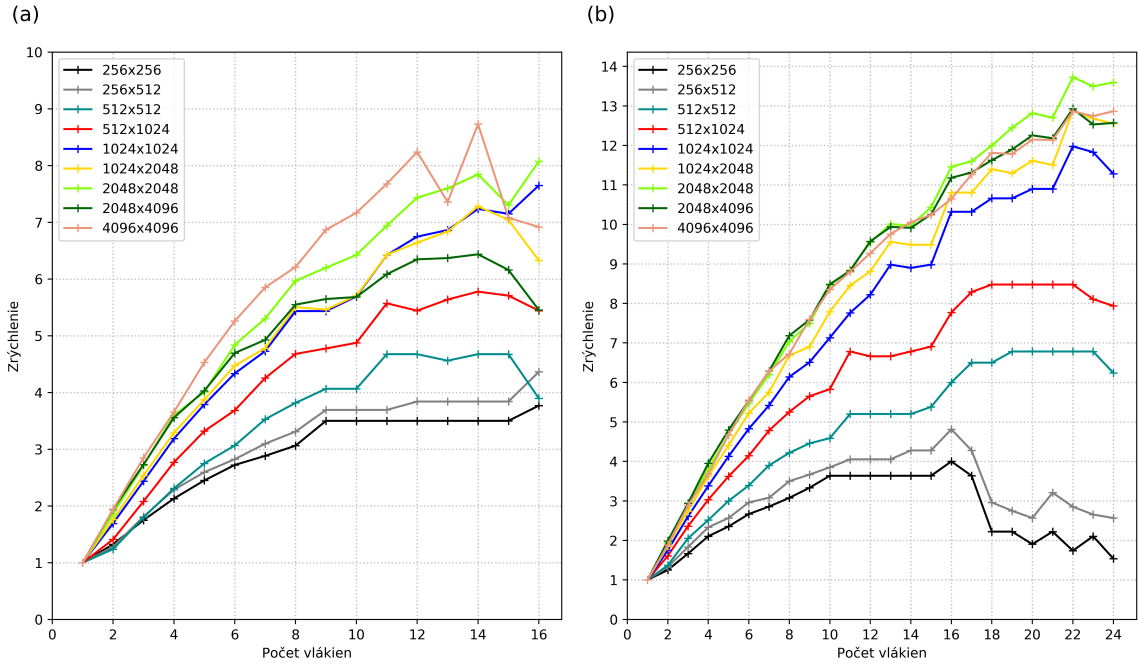


Obr. 8.3: Čas výpočtu jedného kroku simulácie pre rôzne veľkosti 2D domény, finálna 2D/3D implementácia, superpočítač Salomon. Čierna krivka zobrazuje čas výpočtu pre domény, ktorých veľkosti dimenzií sú prvočísla. Sivá krivka zobrazuje čas výpočtu pre domény, ktorých veľkosti dimenzií sú mocniny čísla 2. V grafe sú osi X a Y v logaritmickom merítku.

Obrázok 8.4a (2D doména). Veľkosť mriežky začína na 2^{16} (256x256) bodoch a postupne sa zdvojnásobuje, až na 2^{24} (4096x4096) bodov. Referenčný čas pre každý vstup je čas výpočtu na 1 vlákne. U dvoch najmenších domén je možné pozorovať, že zvyšovanie počtu vlákien nad 8 neprináša výrazné zrýchlenie a je vysoko neefektívne. Réžia spojená s vytváraním a obsluhou veľkého počtu vlákien je vzhľadom na veľkosť domény vysoká. Na obrázku je vidieť, že použitie všetkých 16 vlákien (2xCPU) má zmysel len v prípade veľkosti domény 1024x1024 a 2048x2048. V ostatných prípadoch sa výpočet zrýchli len minimálne, prípadne sa až spomalí. Vo väčšine prípadov je z hľadiska výkonu vhodné použiť 12, prípadne 14 vlákien. Maximálne zrýchlenie bolo dosiahnuté v prípade použitia 14 vlákien, na doméne 4096x4096 (zrýchlenie viac ako 8.5x).

Ďalším experimentálnym meraním je škálovanie výpočtu na superpočítači Salomon uvedené na Obrázku 8.4b. Vstupné dáta boli rovnaké ako v predchádzajúcom prípade, uzol však disponuje viac jadrami (až 24). Zrýchlenie je pre 2 najmenšie domény opäť nevýrazné. Pri použití 16 vlákien sa dostávame na približne štvornásobné zrýchlenie, ktoré ale ďalším zvyšovaním počtu vlákien prudko klesá. Situácia je veľmi podobná ako v predchádzajúcom prípade, maximálne dosiahnuté zrýchlenie je ale výrazne vyššie. V prípade domény 2048x2048 sa za použitia všetkých 24 vlákien vyšplhalo takmer až na 14. V tomto prípade bolo dosiahnutých 58 % z maximálneho, teoretického zrýchlenia (24 jadier – zrýchlenie max. 24 krát). U Anselmu sa táto hodnota vyšplhala na približne 53 %.

Pre overenie škálovania finálnej implementácie pracujúcej nad 3D doménou boli vykonané experimenty zobrazené na Obrázku 8.5. Obrázok 8.5a znova znázorňuje výsledky pre superpočítač Anselm, Obrázok 8.5b pre Salomon. Veľkosť vstupných dát je volená rovnako ako v prípade 2D domény, s rozdielom že v tomto prípade sú body usporiadané do tvaru

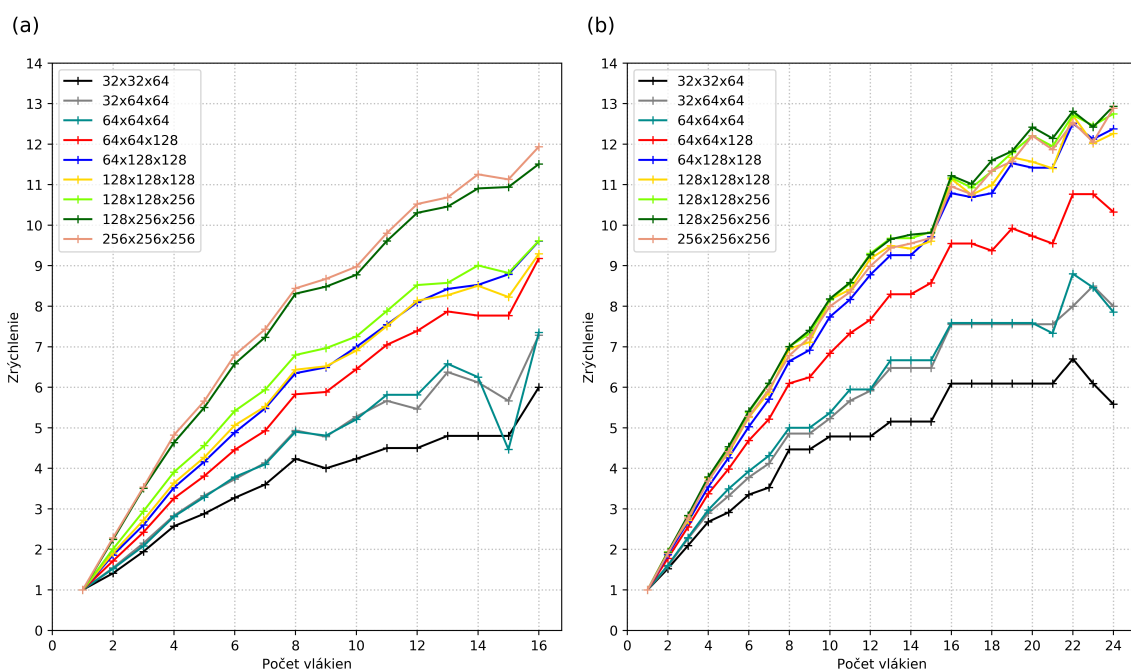


Obr. 8.4: Zrýchlenie výpočtu v závislosti na počte vlákien pre rôzne veľkosti 2D domény, finálna 2D/3D implementácia. (a) superpočítač Anselm, (b) superpočítač Salomon.

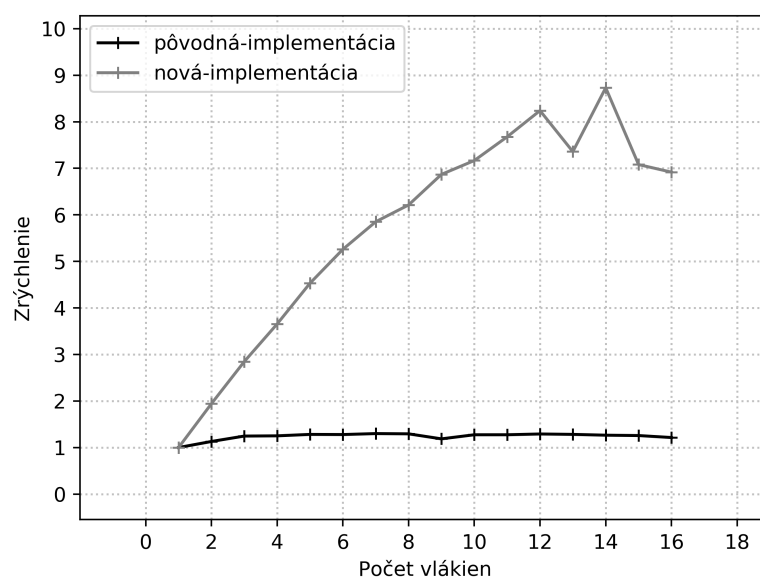
kvádra (napr. $256 * 256 == 32 * 32 * 64 == 2^{16}$ bodov). Výsledky sú tak isto veľmi podobné ako v experimentoch 8.4. Maximálne dosiahnuté zrýchlenie bolo približne 13 pre superpočítač Salomon a 12 pre superpočítač Anselm. Zrýchlenie na superpočítači Anselm je 75 % z maximálneho, teoretického zrýchlenia. Toto maximálne zrýchlenie bolo dosiahnuté pre veľkosť domény 4096x4096 bodov za použitia 16 vlákien, čo značí efektívne škálovanie výpočtu nad dostatočne veľkými vstupnými dátami.

Dôležitým experimentom, ktorý ukazuje význam výslednej implementácie vytvorenej v rámci tejto práce je jej porovnanie s pôvodnou implementáciou 3D k-Wave pracujúcou nad 2D vstupnými dátami. Na Obrázku 8.6 je zobrazené škálovanie pôvodnej a novej implementácie nad 2D doménou s veľkosťou 4096x4096 bodov. Je vidieť, že pôvodná implementácia nieje schopná na 2D vstupných dátach využiť vláknový paralelizmus. Tento fakt je spôsobený najmä tým, že v pôvodnej implementácii sa vykonával paralelný cyklus cez dimenziu Z , ktorej veľkosť je v prípade 2D domény rovná 1. Ďalším dôvodom je vytvorenie *FFT* plánov, ktoré nie sú efektívne pre výpočet 2D *FFT*. Odstránenie týchto zásadných nedostatkov viedlo k výraznému zlepšeniu výkonu pri výpočte v 2D priestore, čo bolo jedným z hlavných cieľov tejto práce.

Ďalším experimentálnym meraním rýchlosti programu bolo spustenie simulácie nad doménou o veľkosti, ktorá odráža potreby reálnych simulácií. V Kapitole 1 bolo spomenuté, že napr. pri transkraniálnej ultrazvukovej neuromodulácii a neurostimulácii sa veľkosť domény môže vyšplhať aj na 16384^2 [12]. Na Obrázku 8.7a je zobrazený priebeh škálovania výpočtu práve nad takto veľkou vstupnou doménou (superpočítač Salomon). Jedná sa o výsledok pre výpočtový uzol z produkčnej fronty (*qprod*, 24 jadier). Zrýchlenie dosiahnuté voči sekvenčnej verzii programu bolo pre tento prípad približne 11. Toto zrýchlenie bolo dosiahnuté za použitia 16 vlákien a ďalej sa už výrazne nezvyšovalo. Priemerná doba výpočtu jedného

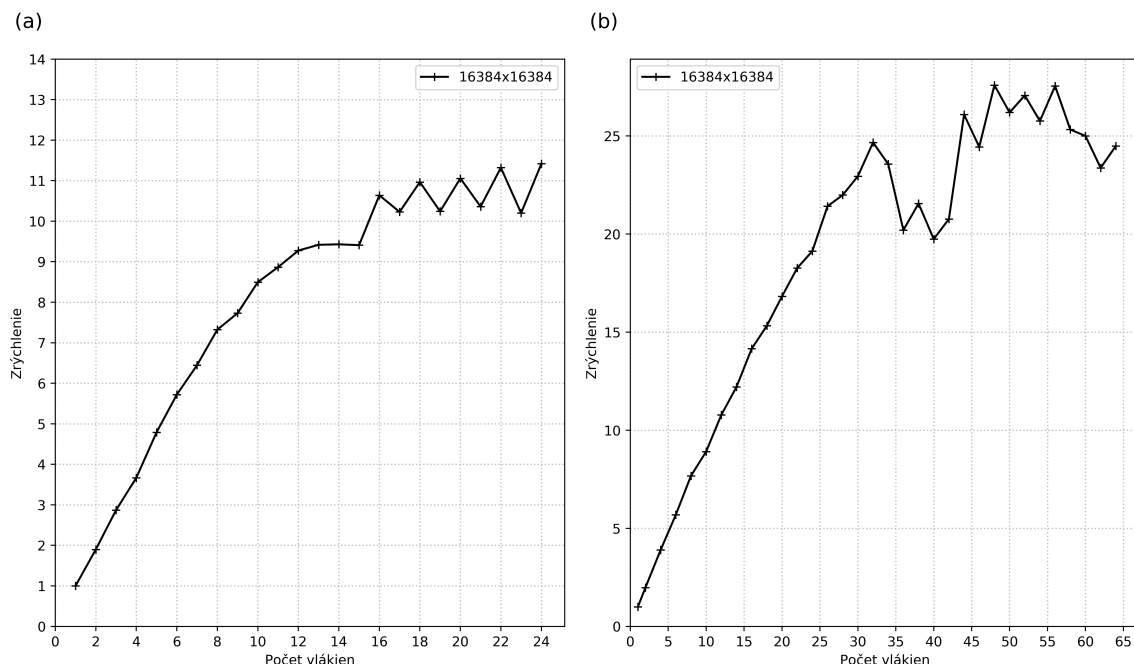


Obr. 8.5: Zrýchlenie výpočtu v závislosti na počte vlákien pre rôzne veľkosti 3D domény, finálna 2D/3D implementácia. (a) superpočítač Anselm, (b) superpočítač Salomon.



Obr. 8.6: Zrýchlenie výpočtu v závislosti na počte vlákien pre 2D doménu o veľkosti 4096x4096 bodov, pôvodná 3D implementácia vs. finálna 2D/3D implementácia, superpočítač Anselm.

roku simulácie bola pre 16 vlákien 2,45 s, pre 24 vlákien 2,32 s. Z tejto hodnoty je možné približne odvodiť výslednú dobu realistickej simulácie podľa počtu potrebných krokov.



Obr. 8.7: Zrýchlenie výpočtu v závislosti na počte vlákien pre 2D doménu o veľkosti 16384x16384, finálna 2D/3D implementácia, superpočítač Salomon. (a) uzol s 24 jadrami (fronta qprod), (b) uzol UV2000 s 112 jadrami (fronta qfat).

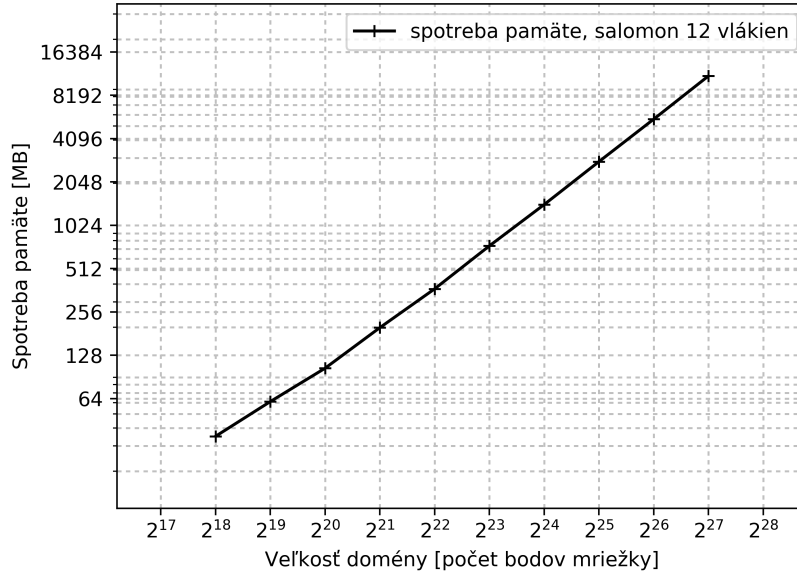
Popísaný experiment bol vykonaný aj na uzle *UV2000* (Obrázok 8.7b), ktorý je osadený až 14 procesormi *Intel Xeon E5-4627v2*, ktoré spolu poskytujú až 112 CPU jadier (fronta *qfat*). Škálovanie výpočtu fungovalo veľmi dobre po hranicu 32 vlákien (4 uzly NUMA), kde bolo dosiahnuté takmer 25 násobné zrýchlenie, čo je 78 % z maximálne možného. Ďalšie pridávanie vlákien (po 44) výpočet spomalilo. Tento fakt mohol byť pravdepodobne spôsobený architektúrou NUMA a spôsobom prepojenia jednotlivých uzlov. Uzly sú prepojené pomocou siete *QPI* liniek v tvare kocky¹. Prepojenie nieje úplné, vzdialenosť medzi alokovanými uzlami môže byť rôzna [13]. V tomto prípade bolo alokovaných 8 NUMA uzlov, teda 64 jadier. Maximálne dosiahnuté zrýchlenie bolo približne 28, za použitia 48 vlákien (6 uzlov). Ďalšie pridávanie jadier nevykazovalo zrýchlenie, preto bol maximálny počet použitých jadier rovný 64.

8.3 Spotreba pamäte

Spotreba pamäte pre testované veľkosti domény je zobrazená na Obrázku 8.8. V tomto prípade bol test vykonaný pre finálnu, zjednotenú implementáciu a 2D vstupné dáta. Spotreba pamäte rastie lineárne s rozmermi domény. Pri zdvojnásobení veľkosti domény sa množstvo spotrebovanej pamäte približne zdvojnásobí. Spotreba operačnej pamäte je približne trojnásobok veľkosti vstupného súboru, v ktorom je uložená vstupná doména. Množstvo

¹Jedná sa o hybridnú 3D kocku, ktorá má navyše prepojené uzly v rámci steny kocky (uhlopriečky) [13]

spotrebovanej pamäte je nezávislé na použitom stroji či type uzlu, preto je uvedený len jeden prípad a síce superpočítač Salomon.

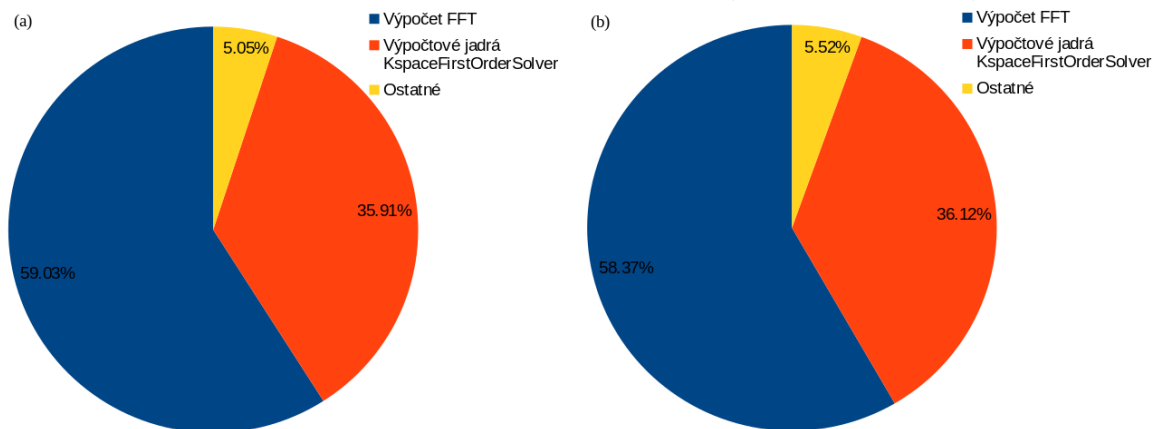


Obr. 8.8: Spotreba pamäte pre rôzne veľkosti 2D domény, finálna 2D/3D implementácia, superpočítač Salomon. V grafe sú osi X a Y v logaritmickom merítku.

8.4 Analýza kritických častí výpočtu

Popri overovaní a meraní výkonu programu je dôležité zistiť, ktoré časti výpočtu sú časovo náročné. Simulácia sa skladá prevažne z výpočtových jadier, ktoré vykonávajú transformáciu vstupných dát na výstupné, pomocou relatívne jednoduchých, aritmetických operácií. Časová zložitosť väčšiny 2D výpočtových jadier, ktoré implementujú nejaký druh transformácie vstupných dát spadá do triedy $t(n) = O(n)$, kde $n = Nx * Ny$. Bodové aritmetické operácie sa v každom kroku vykonávajú totiž cez všetky prvky domény. Podstatné množstvo výpočtových jadier ale navyše volá funkcie pre výpočet *FFT*, ktorej výpočet typicky spadá do triedy lineárne logaritmickej časovej zložitosti ($t(n) = O(n * \log(n))$) [8]. Pre tento prípad je to konkrétne $Nx * Ny * (Nx * \log(Nx)) + Ny * Nx * (Ny * \log(Ny)) + 2(Nx * Ny)$, kde jednotlivé časti výsledného súčtu sú balík 1D *FFT* v smere osi *X*, balík 1D *FFT* v smere osi *Y* a 2x transpozícia.

Na obrázku 8.9 je zobrazený percentuálny podiel jednotlivých častí výpočtu na jeho celkovej dobe. Obrázok 8.9a zobrazuje výsledky meraní, pri ktorých bola na výpočet *FFT* použitá knižnica *Intel MKL*, Obrázok 8.9b zase výsledky pri výpočte *FFT* pomocou knižnice *FFTW*. Výsledky oboch meraní sú veľmi podobné. Celkový čas je rozdelený do 3 častí. Najväčší percentuálny podiel má práve výpočet *FFT* (približne 58 %). Samotné výpočtové jadrá spotrebovávajú asi 36 % času. Zostatok času sa trávi napríklad pri vzorkovaní dát, I/O a iných režijných operáciách.



Obr. 8.9: Percentuálny podiel hlavných častí výpočtu, finálna 2D/3D implementácia, superpočítač Salomon. (a) výpočet FFT pomocou knižnice Intel MKL, (b) výpočet FFT pomocou knižnice FFTW.

Profilovanie programu bolo vykonané pomocou nástroja *Intel VTune Amplifier*². Na tvorbu profilu aplikácie bol použitý program `amplxe-cl`. Vo Výpise 8.1 je uvedený príkaz, ktorý bol spustený pre zozbieranie výsledkov a vytvorenie výslednej správy o meraní.

```

1 # načítanie požadovaného modulu
2 $ ml VTune
3 # spustenie meriania, zozbieranie výsledkov
4 $ amplxe-cl -collect hotspots -result-dir ./r01hs ./kpaceFirstOrder-OMP -i Data/2D/2048
   x2048.h5 -o Data/out.h5 -t 12 --benchmark 1000
5 # vytvorenie výslednej správy o meraní
6 $ amplxe-cl -report hotspots -format csv -limit 50 -r ./r01hs -filter function -show-as
   percent -call-stack-mode all -report-output output.csv

```

Výpis 8.1: Spustenie programu Intel VTune Amplifier za účelom získania profilu implementovaného riešenia

Samotný výpočet (ak neuvažujeme I/O operácie) môže byť limitovaný buď náročnosť vykonávaných operácií, alebo priepustnosťou pamäte. V Tabuľke 8.1 je uvedená maximálna priepustnosť pamäte nameraná pre beh finálnej implementácii na superpočítači Anselm. Experiment bol vykonaný na vstupnej doméne s veľkosťou 4096x4094 za použitie 16 vlákien. Na uzly superpočítača Anselm, ktorý bol použitý pre experiment sa nachádzal procesor *Intel Sandy Bridge E5-2470*, ktorého maximálna pamäťová priepustnosť je 38.4 GB/s³. V uvedenej tabuľke je možné vidieť, že maximálna dosiahnutá priepustnosť pamäte bola 38.2 GB/s, čo je limit uvedeného procesora. Rýchlosť výpočtu je teda silne viazaná na priepustnosť pamäťového subsystému [7].

Meranie priepustnosti pamäte bolo vykonané pomocou programu zo sady nástrojov *Intel PCM*⁴, konkrétne pomocou príkazu `pcm-memory.x` 10. Spustený program monitoruje aktuálnu priepustnosť pamäťového subsystému. Namerané hodnoty následne vypisuje do terminálu, v tomto prípade každých 10 sekúnd. Uvedený experiment nebolo možné vykonať

²<https://software.intel.com/en-us/intel-vtune-amplifier-xe>

³<https://docs.it4i.cz/anselm/compute-nodes/>

⁴<https://software.intel.com/en-us/articles/intel-performance-counter-monitor>

Tabuľka 8.1: Maximálna pamäťová priepustnosť nameraná na superpočítači Anselm, 16 vlákien, finálna 2D/3D implementácia, doména 4096x4094 bodov.

Type merania	Pamäťová priepustnosť [MB/s]
Pamäťová priepustnosť pri čítaní	24505.85
Pamäťová priepustnosť pri zápise	13696.54
Pamäťová priepustnosť celkovo	38202.39

na superpočítači Salomon, nakoľko nieje dostupný modul, ktorý by obsahoval nástroje *Intel PCM*.

8.5 Vplyv frekvencie procesoru na rýchlosť výpočtu

Moderné procesory *Intel* poskytujú možnosť automatickej regulácie frekvencie jednotlivých CPU jadier podľa aktuálnej potreby. Základná frekvencia procesorov *Intel Xeon E5-2680v3*, ktoré sú osadené na väčšine uzlov superpočítača Salomon je 2.5 GHz. Okrem toho využívajú technológiu *Intel Turbo Boost*, ktorá dokáže zvýšiť frekvenciu až na 3.3 GHz (túto funkciu je možné vypnúť). Procesory *Intel Sandy Bridge E5-2665*, ktoré sú osadené na uzloch produkčnej (*qprod*) fronty superpočítača Anselm, majú základnú frekvenciu 2.4 GHz a turbo boost frekvenciu 3.1 GHz. Všetky vykonané experimenty boli spúšťané na uzloch s predvoleným nastavením, v ktorom je turbo boost povolený. V Tabuľke 8.2 je uvedené porovnanie časov simulácie pre prípad povolenej a zakázanej funkcie turbo boost pre superpočítač Anselm. Simulácia (3500 krokov) bola spúšťaná nad doménou 2048x2048 postupne pre 1 až 8 vlákien (8 jadier na jednom sokete). Počas behu programu bola zaznamenávaná aktuálna frekvencia jednotlivých jadier, ktorá bola získaná zo súboru `/proc/cpuinfo`. Priemerná frekvencia (stĺpce frekvencia 1. a frekvencia 2.) bola meraná na jadrách, na ktorých aktuálne bežal spustený program. Program bol naviazaný na konkrétne jadrá CPU pomocou premennej prostredia `OMP_PLACES`.

V Tabuľke 8.2 je vidieť, že v prípade 1. sa frekvencia držala na maximálnej hranici 3.1 GHz pri použití 1 – 3 vlákien. Pre viac ako 4 vlákna začala maximálna frekvencia mierne klesať, pričom sa zastavila na hranici 2.9 GHz. Bolo vykonané aj meranie, pri ktorom bežala simulácia viac ako 100 minút. V tomto prípade klesla frekvencia na 2.7 GHz. Tento fakt je spôsobený výraznejším zahrievaním procesoru v prípade dlhodobej záťaže (v takom prípade CPU automaticky znižuje frekvenciu). Maximálne dosiahnuté zrýchlenie za použitia 8 vlákien bolo 5.75. V meraní číslo 2. (vypnutý turbo boost) sa frekvencia držala na konštantnej hodnote 2.4 GHz. Maximálne dosiahnuté zrýchlenie za použitia 8 vlákien bolo 6.15, čo je vyššia hodnota ako v predchádzajúcom prípade. Tento fakt je spôsobený tým, že procesor pri dlhodobej záťaži všetkých jadier znižuje svoju frekvenciu, čo má za následok zníženie efektivity paralelného výpočtu. To je jeden z dôvodov horšieho škálovania programu na väčšom počte jadier. Tento fakt sa prejavil aj v experimentoch uvedených v predchádzajúcich kapitolách (všetky experimenty boli vykonávané v prostredí s povolenou technológiou turbo boost).

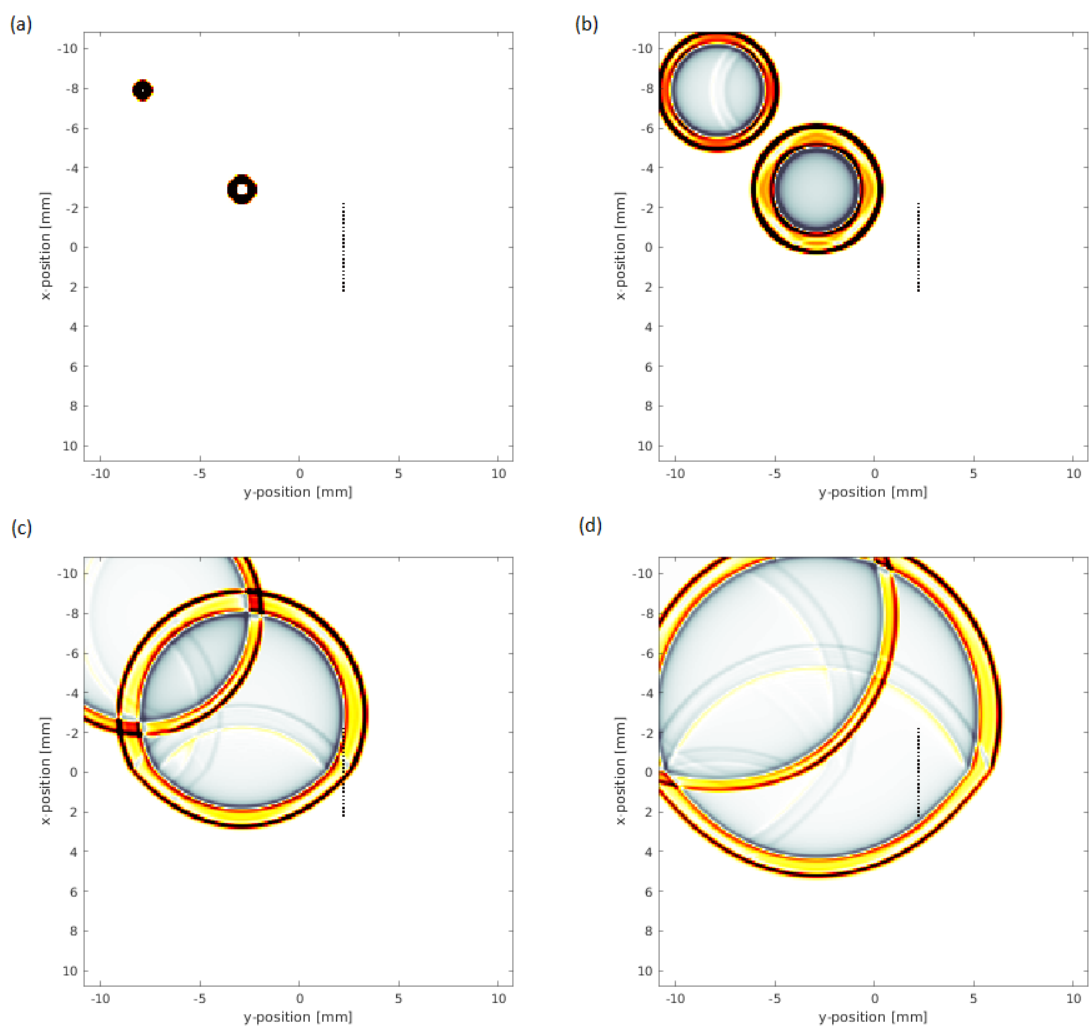
8.6 Vizualizácia výsledkov simulácie

V predchádzajúcich experimentoch boli vstupné dáta simulácie generované pseudonáhodne. Pozorovanými výsledkami boli správnosť výpočtu, jeho rýchlosť či efektivita. V nasledu-

Tabuľka 8.2: Časy výpočtu simulácie na superpočítači Anselm, povolený a zakázaný turbo boost, 1 až 8 vlákien, finálna 2D/3D implementácia, doména 2048x2048 bodov. Čas výpočtu 1. (spolu s frekvenciou 1.) zahŕňa merania pre zapnutý turbo boost, čas výpočtu 2. (spolu s frekvenciou 2.) merania pre vypnutý turbo boost.

Počet vlákien	Čas výpočtu 1. [s]	Frekvencia 1. [s]	Čas výpočtu 2. [MHz/s]	Frekvencia 2. [MHz/s]
1	1035.13	3099	1193.72	2399
2	516.03	3099	603.43	2399
3	363.83	3099	422.42	2399
4	289.82	3080	325.46	2399
5	238.12	3070	266.15	2399
6	208.10	3030	227.94	2399
7	185.48	3000	206.31	2399
8	181.10	2900	194.28	2399

júcom prípade bude výsledok simulácie sledovaný pomocou jej vizualizácie. Vstupná 2D doména bola vytvorená z dvoch médií s rozličnými vlastnosťami (iná hustota, iná rýchlosť šírenia zvuku). Do tejto domény boli umiestnené 2 počiatočné zdroje tlaku. Celá situácia je uvedená na Obrázku 8.10, ktorý je zložený zo 4 snímok zobrazujúcich šírenie akustického tlaku v 2D doméne. Snímky sú označené písmenami (a), (b), (c), (d) a postupne zobrazujú stav simulácie v 1. kroku, 100. kroku, 200. kroku a 300. kroku. Výsledky boli vizualizované pomocou simulačného frameworku *MATLAB* k-Wave.



Obr. 8.10: Simulácia na (pseudo) reálnej 2D doméne, zobrazenie šírenia akustického tlaku. Obrázok (a) – 1. krok simulácie, obrázok (b) – 100. krok simulácie, obrázok (c) – 200. krok simulácie, obrázok (d) – 300. krok simulácie.

Kapitola 9

Testovanie

Pre zachovanie kvality a funkčnosti kódu bol využitý princíp jednotkového a integračného testovania. Pre integračné testy bol využitý existujúci koncept testovania projektu k-Wave. Systém jednotkového testovania bol navrhnutý a implementovaný v rámci tejto práce.

9.1 Jednotkové testy

V *C++ OpenMP* implementácii k-Wave sa doposiaľ nenachádzal žiadny systém jednotkového testovania. Funkčnosť implementácie bola overovaná pomocou sady *MATLAB* skriptov zvanej *kWaveTester*. Počas implementácie zjednoteného 2D a 3D simulačného programu vznikla požiadavka na zavedenie jednotkového testovania, zameraného na overenie funkčnosti implementácie na úrovni jednotlivých výpočtových jadier. Tento krok pomohol overiť funkčnosť novej implementácie a zároveň zaviedol systém, ktorý bude využitý pri ďalšom vývoji simulátoru. Pre tvorbu jednotkových testov bola využitá *C++* knižnica *Google Test*, ktorá bola stručne popísaná v Odseku 5.5.

Základom jednotkových testov k-Wave je trieda `TestCaseSolver` umiestnená v súbore `SolverTests.cpp`. Trieda implementuje 30 jednotkových testov, ktoré overujú funkčnosť najdôležitejších výpočtových jadier tried `KSpaceFirstOrderSolver` a `FftwComplexMatrix`. Jednotlivé testovacie prípady sú implementované pomocou makra `TEST_F()`. Príklad celého jednotkového testu je uvedený vo Výpise C.3 (výpis je kvôli značnému rozsahu umiestnený v Prílohe C). Testovacia trieda pracuje tak, že pred každým testom načíta skalárne hodnoty (veľkosť domény, potrebné konštanty ...) z vstupného *HDF5* súboru (metóda `SetUp`). V každom teste je následne vytvorený kontajner matic, ktorý je naplnený vstupnými hodnotami, načítanými taktiež z vstupného súboru. Okrem toho sú načítané referenčné matice, s ktorými sú porovnávané vypočítané hodnoty. Po načítaní potrebných dát je vytvorená inštancia triedy `KSpaceFirstOrderSolver` a spustená konkrétna metóda, ktorá sa práve testuje. Metóda pracuje s dátami uloženými v kontajnery matic, v ktorom typicky vznikajú aj výstupné hodnoty. Výstupné hodnoty (matice) sú porovnávané s referenčnými maticami. Keďže sa pracuje prevažne s hodnotami typu `float`, tolerovaná odchýlka je $1e-5$. Ak sa vypočítané hodnoty zhodujú s referenčnými, test je prehlásený za úspešný. V opačnom prípade je test neúspešný, pričom je hlásená chyba. Po každom teste je volaná metóda `TearDown`, ktorá uzatvorí vstupný súbor.

Testované metódy sú prevažne implementované ako "chránené", nie je k nim teda možné pristupovať mimo triedy `KSpaceFirstOrderSolver`. Kvôli možnosti testovania chránených a súkromných metód bola testovacia trieda `TestCaseSolver` deklarovaná v rámci

KSpaceFirstOrderSolver ako spriatelena (takzvaná *friend class*). Vo Výpise 9.1 je naznačený spôsob deklarácie spomínanej triedy. Všetky testovacie prípady musia byť deklarované pomocou makra TEST_FRIEND().

```

1 template<bool TIs3d = true>
2 class KSpaceFirstOrderSolver: public KSpaceFirstOrderSolverBase
3 {
4     friend class TestCaseSolver;
5     FRIEND_TEST(TestCaseSolver, tComputeDensityNonlinearScalar);
6     FRIEND_TEST(TestCaseSolver, tComputeDensityLinearScalar);
7     FRIEND_TEST(TestCaseSolver, tComputeVelocityGradient);
8     // ...
9     public:
10         /// Konštruktor
11         KSpaceFirstOrderSolver();
12         // ...
13 };

```

Výpis 9.1: Deklarácia spriatelenej triedy pre účel testovania súkromných metód triedy KSpaceFirstOrderSolver

Vstupné a referenčné dáta sú generované pomocou *MATLAB* skriptu `generateData.m`. V tomto skripte sú najskôr nastavené potrebné konštanty (napr. veľkosť domény), podľa ktorých sú vygenerované pseudonáhodné vstupné dáta (matice). Dáta následne vstupujú do *MATLAB* funkcií, ktoré vykonávajú rovnaký výpočet, ako *C++* výpočtové jadrá (*C++* kód je implementovaný podľa referenčného *MATLAB* kódu). Skalárne konštanty, vstupné a referenčné matice sú uložené do spomínaného *HDF5* súboru, ktorý je vstupom pre testy KSpaceFirstOrderSolver. Každý test je v *HDF5* súbore uložený ako samostatná skupina (*HDF5 group*), čo umožňuje uložiť dáta všetkých testov do 1 súboru.

Pre preklad k-Wave spolu s jednotkovými testami je nutné, aby bol na danom prostredí nainštalovaný framework *Google Test*. Postup inštalácie tejto knižnice (pre Ubuntu-16.04, server *sc-gpu1.fit.vutbr.cz*) je uvedený v prílohe C. Táto príloha obsahuje aj postup kompilácie testov k-Wave spolu s ich spustením.

9.2 Integračné testy

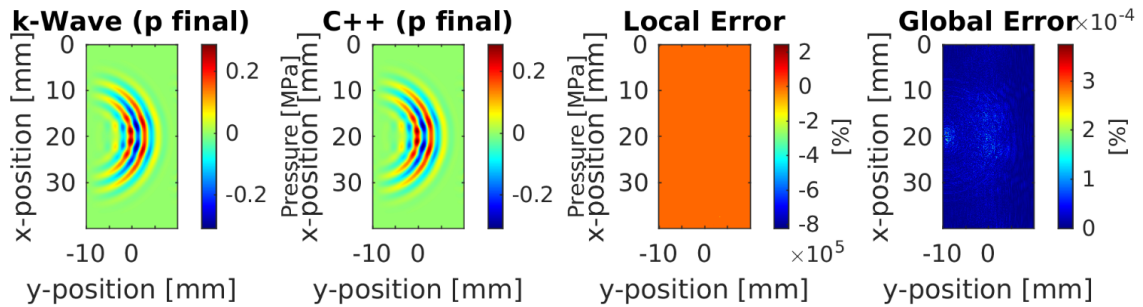
Pre účeli integračného testovania bol využitý testovací framework *kWaveTester*, ktorý bol vyvinutý pre testovanie pôvodnej *C++* implementácie k-Wave. Tento framework je vytvorený v prostredí *MATLAB*.

Framework *kWaveTester* automatizuje proces integračného testovania k-Wave. Zabezpečuje generovanie vstupných dát, nastavuje parametre simulácie, spúšťa testovaný program, porovnáva, vyhodnocuje a vizualizuje výsledky. Referenčné výsledky získava z *MATLAB* k-Wave a porovnáva ich s výsledkami testovaného programu. Framework teda testuje a porovnáva výsledky celej simulácie (nad vstupným súborom sa spustí referenčná a testovaná implementácia). Vykonáva sa relatívne veľký počet krokov simulácie, čo môže odhaliť kumulované chyby. Na konci testu je vypočítaná chyba (odchýlka) medzi testovaným a referenčným riešením. Tolerovaná odchýlka je $1e-5$.

MATLAB k-Wave je implementovaná pre 1D, 2D a 3D priestor, čo umožňuje testovanie novej 2D *C++* implementácie. Framework nemusí nutne porovnávať výsledky testovaného programu, je možné ho využiť napr. na generovanie vstupných dát. Tester obsahuje sadu

definovaných testov, ktoré pokrývajú mnohé prípady použitia. Okrem toho je možné vytvoriť vlastné testovacie prípady. U jednotlivých testov je možné definovať veľkosť domény, jej dimensionalitu, vlastnosti média, vlastnosti akustickej vlny, možnosť vizualizácie výsledkov apod.

Pre účely testovania novej implementácie bol vytvorená sada integračných testov, ktoré sú spúšťané pomocou frameworku *kWaveTester*. Sada testov je implementovaná v skripte `kwt_run_OMP_medium_2D.m` (adresár `Testing/kWaveTester`). Integračné testy boli rovnako ako aj ostatné experimenty spúšťané nad rôzne veľkými vstupmi. Maximálna chyba novej implementácie voči *MATLAB* k-Wave je aj v tomto prípade $1e-5$. Framework *kWaveTester* umožňuje aj vizualizáciu výstupov testovaného riešenia spolu so zobrazením chyby. Na Obrázku 9.1 je zobrazený príklad vizualizácie výstupu integračných testov. Konkrétne sa jedná o zobrazenie finálneho tlaku vypočítaného pomocou *MATLAB* k-Wave a *C++* k-Wave (testované riešenie), ďalej zobrazenie odchýlky (chyby) medzi jednotlivými riešeniami.



Obr. 9.1: Vizualizácia výstupov integračného testu (finálny tlak). Obrázky zľava doprava zobrazujú finálny tlak vypočítaný pomocou *MATLAB* k-Wave, finálny tlak vypočítaný pomocou testovaného riešenia (2D/3D implementácia), lokálnu a globálnu chybu.

Kapitola 10

Záver

Simulačný nástroj k-Wave doposiaľ implementoval 2D simuláciu ultrazvuku pomocou *MATLAB* frameworku. Tento spôsob ale nebol z časového a ekonomického hľadiska vhodný pre riešenie reálnych simulácií veľkých rozmerov (napr. 16384^2). Nakoľko simulácia uvedených rozmerov môže trvať aj niekoľko dní, je nutné hľadať riešenia, ktoré poskytnú požadovaný výsledok, pričom sú časovo a finančne prijateľnejšie. V rámci diplomovej práce teda prebehlo zoznámenie sa so simulačným programom k-Wave a jeho akcelerovanou 3D implementáciou určenou pre superpočítačové systémy. Na základe referenčnej 2D *MATLAB* implementácie a akcelerovanej 3D verzie programu bola vytvorená nová, 2D simulácia implementovaná v jazyku *C++*, akcelerovaná pomocou *OpenMP*.

Prototyp 2D ultrazvukovej simulácie je implementovaný ako samostatný program a je určený pre stroje s architektúrou zdieľanej pamäte. Pracuje nad 2D vstupnými doménami, pričom dôraz je kladený na rýchlosť a efektivitu výpočtu. Výkon a efektivita programu bola experimentálne overená na rozsiahlej sade testov. Zámerom experimentov bolo vyhodnotiť čas výpočtu jedného kroku simulácie pre rôzne veľké domény a rôzne konfigurácie strojov. Okrem toho prebehlo meranie zrýchlenia voči sekvenčnej verzii programu. Maximálne dosiahnuté zrýchlenie bolo viac ako 8 pre superpočítač Anselm a 14 pre superpočítač Salomon.

Jedným z hlavných cieľov diplomovej práce bol návrh a implementácia simulačného programu, ktorý zjednocuje výpočty nad 2D a 3D doménami. Dôraz je kladený jednak na rýchlosť a efektivitu výpočtu, ale aj na prehľadnosť a štruktúru kódu. Pre dosiahnutie uvedených vlastností boli použité moderné prostriedky *C++*, generické programovanie, *OpenMP* apod. Výkon a efektivita implementácie bola overená na veľkom množstve experimentov. Opäť bol meraný čas výpočtu jedného kroku simulácie pre rôzne veľké domény a škálovanie výpočtu. Nová, zjednotená 2D/3D implementácia zachováva nad 2D doménami výkonové vlastnosti prototypu 2D simulácie a nad 3D doménami vlastnosti pôvodnej 3D *C++* verzie, pričom zjednocuje kódy oboch implementácií do jedného, prehľadného celku. Maximálne zrýchlenie novej, paralelnej implementácie (akcelerácia pomocou *OpenMP*) je voči sekvenčnej verzii 8 pre superpočítač Anselm a 14 pre superpočítač Salomon. Rýchlosť vytvoreného riešenia bola meraná aj pre simuláciu nad vstupnou doménou značnej veľkosti, konkrétne 16384^2 bodov. Takáto veľkosť vstupu môže byť vyžadovaná pre dosiahnutie presných výsledkov rôznych realistických simulácií. V uvedenom prípade bolo dosiahnuté takmer jedenásťnásobné zrýchlenie voči sekvenčnej verzii programu za použitia 16 vlákien (superpočítač Salomon). Priemerná doba výpočtu jedného kroku simulácie bola 2.48 s.

Jedným z najdôležitejších meraní bolo porovnanie výkonu novej, akcelerovanej implementácie voči *MATLAB* verzii k-Wave. Zistilo sa, že napr. pre doménu 4096^2 dosahuje

nová implementácia voči *MATLAB* k-Wave sedemnásobné zrýchlenie na superpočítači Anselm a osemnásobné zrýchlenie na superpočítači Salomon. Tento fakt výrazne znižuje cenu simulácie pri zachovaní požadovanej presnosti. Pre potencionálneho užívateľa k-Wave má rozhodne zmysel okrem inštalácie *MATLAB* frameworku aj inštalácia nového, akcelerovaného riešenia. Maximálna nameraná a tolerovaná chyba novej implementácie v porovnaní s referenčnou, *MATLAB* k-Wave dosahuje hodnotu $1e-5$.

Testy pre Anselm a Salomon na jednom jadre ukázali vplyv inštrukčnej sady *AVX2* na celkovú rýchlosť výpočtu. Čas výpočtu na jednom jadre uzla Salomon (*Intel Xeon E5-2680v3*, *Haswell*, *AVX2*) bol približne o 20 % nižší ako na jednom jadre uzla Anselm (*Intel Xeon E5-2665*, *Sandy Bridge*, *AVX*). Je treba dodať, že prvý z uvedených procesorov pracuje na mierne vyššej frekvencii (o 100 až 200 MHz), takže zrýchlenie je ovplyvnené aj týmto faktorom. Pri použití novej generácii procesorov *Intel Skylake-X*, (*AVX512*) sa dá predpokladať zrýchlenie, ktoré však môže byť limitované priepustnosťou pamäťového subsystému.

Z grafov škálovania je možné vyvodiť záver, že nemá veľký zmysel používať procesory s príliš veľkým počtom jadier (najmä pre malé domény). Grafy ukazujú, že pridávanie jadier má zmysel, ak zvyšujeme veľkosť riešeného problému (slabé škálovanie). Výpočet väčšinou veľmi slušne škáluje s pridávaním jadier po hodnotu 8, kde sa dosahuje asi sedemnásobné zrýchlenie. Ďalšie pridávanie jadier je už menej efektívne. Kritická je priepustnosť pamäte. Pri zostavovaní stroja určeného pre beh vytvorenej implementácie by mohlo byť vhodné použiť napr. osem jadrový procesor s frekvenciou nad 3.0 GHz a rýchle 4 kanálové pamäte. Značný vplyv na výkon má aj technológia *Intel Turbo Boost*, ktorá v prípade potreby zvyšuje frekvenciu jadier CPU. V prípade dlhodobého zataženia všetkých jadier sa však frekvencia znižuje, čo má za následok horšie škálovanie výpočtu na veľkom počte jadier.

Analýzou kritických miest programu sa zistilo, že asi 60 % času sa trávi pri výpočte *FFT*. Nakoľko sa pre výpočet *FFT* používajú špecializované a vysoko optimalizované knižnice, nieje už veľký priestor na ďalšiu optimalizáciu a zvyšovanie výkonu. Teoretickou možnosťou by bola zmena metódy výpočtu na pseudospektrálnu, to by ale viedlo k nutnosti zvýšenia rozlíšenia a navýšenia počtu simulačných krokov. Ďalším meraním sa zistilo, že rýchlosť výpočtu je silne viazaná na priepustnosť pamäťového subsystému. Na uzle superpočítača Anselm bola nameraná priepustnosť 38,2 GB/s, čo je limit daného procesora.

Pre finálnu implementáciu bola navrhnutá a vytvorená sada jednotkových testov, ktoré overujú funkčnosť programu na úrovni jednotlivých výpočtových jadier. Navrhnutý koncept jednotkového testovania bude možné v projekte *k-Wave-Fluid-OMP* ďalej rozširovať, prípadne ho bude možné využiť aj v ďalších k-Wave projektoch. Projekt je momentálne pripravený na koncept vývoja riadeného testami (*Test Driven Development*). Pri implementácii nových výpočtových jadier už bude jednoduché pridať príslušný jednotkový test do existujúceho frameworku. Implementované riešenie bolo okrem toho testované aj na úrovni integračných testov pomocou *MATLAB* frameworku *kWaveTester*.

Vytvorené riešenie zjednotenej 2D/3D simulácie bude možné v budúcnosti rozširovať a integrovať aj v iných projektoch k-Wave. Existujú ďalšie implementácie k-Wave určené napríklad pre systémy s akcelerátormi GPU či zväzky počítačov (cluster), ktoré dnes pracujú nad 3D doménami. V prípade potreby by bolo možné na základe vytvorenej, zjednotenej 2D/3D simulácie implementovať výpočty nad 2D doménami aj do ďalších uvedených verzií k-Wave.

Literatúra

- [1] Al-Bataineh, O.; Jenne, J.; Huber, P.: Clinical and future applications of high intensity focused ultrasound in cancer. *Cancer Treatment Reviews*, ročník 38, č. 5, 2012: s. 346–353, ISSN 0305-7372, doi:10.1016/j.ctrv.2011.08.004.
- [2] Architecture, O.; Board, R.: The OpenMP API specification for parallel programming. 2015: str. 359.
URL <http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf>
- [3] Crichlow, J.: *An Introduction to Distributed and Parallel Computing*. Prentice Hall, 1997, ISBN 9780131909687.
- [4] Dvořák, V. a Jaroš, J.: *Študijné materiály k predmetu Architektúra a programovanie paralelných systémov*. Vysoké učení technické v Brně, Fakulta informačních technologií., [Online; navštívené 17.12.2017].
- [5] Harstell, W. F.; Scott, C. B.; Bruner, D. W.; aj.: Randomized trial of short- versus long-course radiotherapy for palliation of painful bone metastases. *Journal of the National Cancer Institute*, ročník 97, č. 11, 2005: s. 798–804, ISSN 1460-2105, doi:10.1093/jnci/dji139.
- [6] *Dokumentácia IT4Innovations pre superpočítače Anselm a Salomon*. [Online; navštíveno 20.12.2016].
URL <https://docs.it4i.cz/>
- [7] Jaroš, J.: *High Performance Computing in Ultrasound Cancer Treatment*. Habilitation thesis, Brno Univeristy of Technology, Brno, 2017.
URL <http://www.fit.vutbr.cz/research/habilitace/index.php.cs?id=11539&type=HABIL>
- [8] Lohne, M.: The Computational Complexity of the Fast Fourier Transform. 2017: s. 10–12, [Online; navštívené 29.04.2018].
URL <https://folk.uio.no/mathialo/texts/fftcomplexity.pdf>
- [9] Meyers, S.: *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. O'Reilly, 2014, ISBN 9781491903995.
- [10] van der Pas, R.; Stotzer, E.; Terboven, C.: *Using OpenMP – The Next Step: Affinity, Accelerators, Tasking, and SIMD*. Scientific and Engineering Computation, MIT Press, 2017, ISBN 9780262344029.
- [11] Prata, S.; Sokol, B.: *Mistrovství v C++*. Bestseller (Computer Press), Computer Press, 2013, ISBN 9788025138281.

- [12] Robertson, J. L. B.; Cox, B. T.; Jaros, J.; aj.: Accurate simulation of transcranial ultrasound propagation for ultrasonic neuromodulation and stimulation. *The Journal of the Acoustical Society of America*, ročník 141, č. 3, mar 2017: s. 1726–1738, ISSN 0001-4966, doi:10.1121/1.4976339.
- [13] *SGI UV 2000 System User Guide*. 2012.
URL <http://irix7.com/techpubs/007-5832-001.pdf>
- [14] Shen, H.; Pétrot, F.: Using Amdahl's law for performance analysis of many-core SoC Architectures based on functionally asymmetric processors. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, ročník 6566 LNCS, č. February 2011, 2011: s. 38–49, ISSN 03029743, doi:10.1007/978-3-642-19137-4_4.
- [15] Suomi, V.; Jaros, J.; Treeby, B. E.; aj.: Nonlinear 3-D simulation of high-intensity focused ultrasound therapy in the Kidney. In *2016 38th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, IEEE, aug 2016, ISSN 1557-170X, s. 5648–5651, doi:10.1109/EMBC.2016.7592008.
- [16] Treeby, B. E.; Cox, B. T.: k-Wave: MATLAB toolbox for the simulation and reconstruction of photoacoustic wave fields. *Journal of Biomedical Optics*, ročník 15, č. 2, 2010: str. 021314, ISSN 10833668, doi:10.1117/1.3360308.
- [17] Treeby, B. E.; Jaros, J.; Rendell, A. P.; aj.: Modeling nonlinear ultrasound propagation in heterogeneous media with power law absorption using a k-space pseudospectral method. *The Journal of the Acoustical Society of America*, ročník 131, č. 6, 2012: s. 4324–36, ISSN 1520-8524, doi:10.1121/1.4712021.
- [18] Treeby, B. E.; Jaros, J.; Rohrbach, D.; aj.: Modelling Elastic Wave Propagation Using the k-Wave MATLAB Toolbox. *IEEE International Ultrasonics Symposium*, , č. 4, 2014.
- [19] Vandevoorde, D.; Josuttis, N.; Gregor, D.: *C++ Templates: The Complete Guide*. Addison Wesley Professional, 2017, ISBN 9780321714121.
- [20] *World Health Organization*. [Online; navštívené 20.12.2017].
URL <http://www.who.int/mediacentre/factsheets/fs297/en/>

Príloha A

Prepínače programu kspaceFirstOrder3D-OMP

1	+-----+-----+-----+		
2		kspaceFirstOrder3D-OMP v1.2	
3	+-----+-----+-----+		
4		Usage	
5	+-----+-----+-----+		
6		Mandatory parameters	
7	+-----+-----+-----+		
8		-i <file_name>	HDF5 input file
9		-o <file_name>	HDF5 output file
10	+-----+-----+-----+		
11		Optional parameters	
12	+-----+-----+-----+		
13		-t <num_threads>	Number of CPU threads
14			(default = 4)
15		-r <interval_in_%>	Progress print interval
16			(default = 5%)
17		-c <compression_level>	Compression level <0,9>
18			(default = 0)
19		--benchmark <time_steps>	Run only a specified number
20			of time steps
21		--verbose <level>	Level of verbosity <0,2>
22			0 - basic, 1 - advanced,
23			2 - full
24			(default = basic)
25		-h, --help	Print help
26		--version	Print version and build info
27	+-----+-----+-----+		
28		--checkpoint_file <file_name>	HDF5 checkpoint file
29		--checkpoint_interval <sec>	Checkpoint after a given
30			number of seconds
31	+-----+-----+-----+		
32		Output flags	
33	+-----+-----+-----+		
34		-p	Store acoustic pressure
35			(default output flag)
36			(the same as --p_raw)
37		--p_raw	Store raw time series of p
38		--p_rms	Store rms of p
39		--p_max	Store max of p
40		--p_min	Store min of p

```

41 | --p_max_all          | Store max of p (whole domain) |
42 | --p_min_all          | Store min of p (whole domain) |
43 | --p_final            | Store final pressure field    |
44 +-----+-----+
45 | -u                  | Store ux, uy, uz             |
46 |                    | (the same as --u_raw)        |
47 | --u_raw             | Store raw time series of      |
48 |                    | ux, uy, uz                   |
49 | --u_non_staggered_raw | Store non-staggered raw time |
50 |                    | series of ux, uy, uz         |
51 | --u_rms             | Store rms of ux, uy, uz       |
52 | --u_max             | Store max of ux, uy, uz       |
53 | --u_min             | Store min of ux, uy, uz       |
54 | --u_max_all         | Store max of ux, uy, uz       |
55 |                    | (whole domain)               |
56 | --u_min_all         | Store min of ux, uy, uz       |
57 |                    | (whole domain)               |
58 | --u_final           | Store final acoustic velocity |
59 +-----+-----+
60 | -s <time_step>      | When data collection begins   |
61 |                    | (default = 1)                 |
62 +-----+-----+

```

Výpis A.1: Výpis nápovedy programu, možné parametre programu kspaceFirstOrder3D-OMP zadané do príkazového riadku

Príloha B

Príklad výstupu programu kspaceFirstOrder3D-OMP

```
1 $ ml HDF5/1.8.18-intel-2017a-serial
2 $ ./kspaceFirstOrder3D-OMP-ref -i Data/input_data_128_128_128_het_index.h5 -o Data/out.h5 -
   t 4 -p
3 +-----+
4 |               kspaceFirstOrder3D-OMP v1.2               |
5 +-----+
6 | Reading simulation configuration:                           Done |
7 | Number of CPU threads:                                     4 |
8 +-----+
9 |               Simulation details                           |
10 +-----+
11 | Domain dimensions:                                     128 x 128 x 128 |
12 | Simulation time steps:                                   321 |
13 +-----+
14 |               Initialization                               |
15 +-----+
16 | Memory allocation:                                       Done |
17 | Data loading:                                           Done |
18 | Elapsed time:                                           0.16s |
19 +-----+
20 | FFT plans creation:                                       Done |
21 | Pre-processing phase:                                    Done |
22 | Elapsed time:                                           0.07s |
23 +-----+
24 |               Computational resources                     |
25 +-----+
26 | Current host memory in use:                             216MB |
27 +-----+
28 |               Simulation                                  |
29 +-----+
30 | Progress | Elapsed time | Time to go | Est. finish time |
31 +-----+
32 | 0% | 0.415s | 66.235s | 12/05/18 17:44:02 |
33 | 5% | 7.212s | 121.395s | 12/05/18 17:45:04 |
34 | 10% | 14.981s | 126.459s | 12/05/18 17:45:17 |
35 | 15% | 19.938s | 108.062s | 12/05/18 17:45:04 |
36 | 20% | 24.946s | 96.382s | 12/05/18 17:44:57 |
37 | 25% | 31.288s | 91.192s | 12/05/18 17:44:58 |
38 | 30% | 37.072s | 84.357s | 12/05/18 17:44:57 |
39 | 35% | 41.591s | 75.520s | 12/05/18 17:44:53 |
```

```

40 | 40% | 46.177s | 67.845s | 12/05/18 17:44:49 |
41 | 45% | 56.540s | 67.771s | 12/05/18 17:45:00 |
42 | 50% | 62.656s | 61.495s | 12/05/18 17:45:00 |
43 | 55% | 69.763s | 56.045s | 12/05/18 17:45:02 |
44 | 60% | 77.802s | 50.932s | 12/05/18 17:45:04 |
45 | 65% | 83.314s | 44.037s | 12/05/18 17:45:03 |
46 | 70% | 88.658s | 37.268s | 12/05/18 17:45:02 |
47 | 75% | 95.056s | 31.031s | 12/05/18 17:45:02 |
48 | 80% | 98.010s | 23.933s | 12/05/18 17:44:57 |
49 | 85% | 103.126s | 17.690s | 12/05/18 17:44:56 |
50 | 90% | 113.255s | 12.107s | 12/05/18 17:45:01 |
51 | 95% | 119.372s | 5.852s | 12/05/18 17:45:00 |
52 +-----+-----+-----+-----+
53 | Elapsed time: 125.48s |
54 +-----+-----+-----+-----+
55 | Sampled data post-processing: Done |
56 | Elapsed time: 0.02s |
57 +-----+-----+-----+-----+
58 | Summary |
59 +-----+-----+-----+-----+
60 | Peak memory in use: 218MB |
61 +-----+-----+-----+-----+
62 | Total execution time: 125.83s |
63 +-----+-----+-----+-----+
64 | End of computation |
65 +-----+-----+-----+-----+

```

Výpis B.1: Príklad spustenia a výstupu programu kspaceFirstOrder3D-OMP

Príloha C

Jednotkové testy

```
1 GTEST_INSTALL_PATH="/home/xsimek23/gtest"
2 wget https://github.com/google/googletest/archive/master.zip
3 unzip master.zip
4 cd googletest-master/
5 mkdir build
6 cd build
7
8 cmake ../ -DCMAKE_INSTALL_PREFIX="$GTEST_INSTALL_PATH"
9 make
10 make install
```

Výpis C.1: Preklad a inštalácia knižnice Google tests

Po inštalácii je nutné nastaviť správnu cestu (premenná `GTEST_DIR`) ku knižnici v súbore `Makefiles/Ubuntu-16.04/Makefile`. Na serveri *sc-gpu1.fit.vutbr.cz* je možné využiť už nainštalovaný modul pomocou príkazu `ml googletest`.

```
1 # kompilácia jednotkových testov
2 [xsimek23@sc-gpu1 k-Wave-Fluid-OMP-2D]$ make -f Makefiles/Ubuntu-16.04/Makefile test
3 # spustenie jednotkových testov
4 [xsimek23@sc-gpu1 k-Wave-Fluid-OMP-2D]$ ./kSpaceFirstOrder-OMP-test
5 [=====] Running 30 tests from 1 test case.
6 [-----] Global test environment set-up.
7 [-----] 30 tests from TestCaseSolver
8 [ RUN      ] TestCaseSolver.tComputeDensityNonlinearScalar
9 [          OK ] TestCaseSolver.tComputeDensityNonlinearScalar (37 ms)
10 [ RUN      ] TestCaseSolver.tComputeDensityNonlinearMatrix
11 [          OK ] TestCaseSolver.tComputeDensityNonlinearMatrix (5 ms)
12 [ RUN      ] TestCaseSolver.tComputeDensityLinearScalar
13 [          OK ] TestCaseSolver.tComputeDensityLinearScalar (4 ms)
14 [ RUN      ] TestCaseSolver.tComputeDensityLinearMatrix
15 [          OK ] TestCaseSolver.tComputeDensityLinearMatrix (3 ms)
16 [ RUN      ] TestCaseSolver.tComputeVelocityGradient
17 [          OK ] TestCaseSolver.tComputeVelocityGradient (230 ms)
18 [ RUN      ] TestCaseSolver.tFft3D
19 [          OK ] TestCaseSolver.tFft3D (2 ms)
20 [ RUN      ] TestCaseSolver.tFft2D
21 [          OK ] TestCaseSolver.tFft2D (54 ms)
22 [ RUN      ] TestCaseSolver.tFft3DShift
23 [          OK ] TestCaseSolver.tFft3DShift (35 ms)
24 [ RUN      ] TestCaseSolver.tFft2DShift
25 [          OK ] TestCaseSolver.tFft2DShift (26 ms)
26 [ RUN      ] TestCaseSolver.tComputeShiftedVelocity
```

```

27 [      OK ] TestCaseSolver.tComputeShiftedVelocity (6 ms)
28 [ RUN      ] TestCaseSolver.tSumPressureTermsLinearLosslessScalar
29 [      OK ] TestCaseSolver.tSumPressureTermsLinearLosslessScalar (1 ms)
30 [ RUN      ] TestCaseSolver.tSumPressureTermsLinearLosslessMatrix
31 [      OK ] TestCaseSolver.tSumPressureTermsLinearLosslessMatrix (3 ms)
32 [ RUN      ] TestCaseSolver.tSumPressureTermsNonlinearLosslessScalar
33 [      OK ] TestCaseSolver.tSumPressureTermsNonlinearLosslessScalar (2 ms)
34 [ RUN      ] TestCaseSolver.tSumPressureTermsNonlinearLosslessMatrix
35 [      OK ] TestCaseSolver.tSumPressureTermsNonlinearLosslessMatrix (2 ms)
36 [ RUN      ] TestCaseSolver.tComputePressureGradient
37 [      OK ] TestCaseSolver.tComputePressureGradient (6 ms)
38 [ RUN      ] TestCaseSolver.tComputeVelocityHomogeneousUniform
39 [      OK ] TestCaseSolver.tComputeVelocityHomogeneousUniform (20 ms)
40 [ RUN      ] TestCaseSolver.tComputeVelocityHomogeneousNonuniform
41 [      OK ] TestCaseSolver.tComputeVelocityHomogeneousNonuniform (6 ms)
42 [ RUN      ] TestCaseSolver.tComputeVelocityHeterogenous
43 [      OK ] TestCaseSolver.tComputeVelocityHeterogenous (6 ms)
44 [ RUN      ] TestCaseSolver.tGenerateInitialDenisty
45 [      OK ] TestCaseSolver.tGenerateInitialDenisty (3 ms)
46 [ RUN      ] TestCaseSolver.tComputeInitialVelocityHeterogeneous
47 [      OK ] TestCaseSolver.tComputeInitialVelocityHeterogeneous (11 ms)
48 [ RUN      ] TestCaseSolver.tComputeInitialVelocityHomogeneousUniform
49 [      OK ] TestCaseSolver.tComputeInitialVelocityHomogeneousUniform (14 ms)
50 [ RUN      ] TestCaseSolver.tComputeInitialVelocityHomogeneousNonuniform
51 [      OK ] TestCaseSolver.tComputeInitialVelocityHomogeneousNonuniform (5 ms)
52 [ RUN      ] TestCaseSolver.tComputePressureTermsLinearScalar
53 [      OK ] TestCaseSolver.tComputePressureTermsLinearScalar (3 ms)
54 [ RUN      ] TestCaseSolver.tComputePressureTermsLinearMatrix
55 [      OK ] TestCaseSolver.tComputePressureTermsLinearMatrix (3 ms)
56 [ RUN      ] TestCaseSolver.tComputePressureTermsNonlinearScalar
57 [      OK ] TestCaseSolver.tComputePressureTermsNonlinearScalar (3 ms)
58 [ RUN      ] TestCaseSolver.tComputePressureTermsNonlinearMatrix
59 [      OK ] TestCaseSolver.tComputePressureTermsNonlinearMatrix (10 ms)
60 [ RUN      ] TestCaseSolver.tComputeAbsorbtionTerm
61 [      OK ] TestCaseSolver.tComputeAbsorbtionTerm (13 ms)
62 [ RUN      ] TestCaseSolver.tGenerateTauAndEta
63 [      OK ] TestCaseSolver.tGenerateTauAndEta (9 ms)
64 [ RUN      ] TestCaseSolver.tGenerateKappa
65 [      OK ] TestCaseSolver.tGenerateKappa (3 ms)
66 [ RUN      ] TestCaseSolver.tGenerateKappaAndNablas
67 [      OK ] TestCaseSolver.tGenerateKappaAndNablas (3 ms)
68 [-----] 30 tests from TestCaseSolver (529 ms total)
69
70 [-----] Global test environment tear-down
71 [=====] 30 tests from 1 test case ran. (529 ms total)
72 [ PASSED ] 30 tests.

```

Výpis C.2: Preklad a spustenie jednotkových testov programu kspaceFirstOrder-OMP

```

1 // Test metódy ComputeVelocityGradient
2 TEST_F(TestCaseSolver, tComputeVelocityGradient)
3 {
4     // príprava prostredia testu, volanie metódy SetUp,
5     // ktorá načíta skarálne hodnoty zo vstupného súboru
6     std::string rootGroup = kRootGroup0;
7     std::string testGroup = rootGroup + "/compVelocityGradient";
8     SetUp(rootGroup, kDataInputFile);
9     Hdf5File& inputFile = mParams.getInputFile();

```

```

10 // vytvorenie a načítanie referenčných matíc
11 DimensionSizes fullDims = mParams.getFullDimensionSizes();
12 DimensionSizes reducedDims = mParams.getReducedDimensionSizes();
13 RealMatrix duxdxRef(fullDims);
14 RealMatrix duydyRef(fullDims);
15 RealMatrix duzdzRef(fullDims);
16 duxdxRef.readData(inputFile, testGroup+"/duxdx_ref");
17 duydyRef.readData(inputFile, testGroup+"/duydy_ref");
18 duzdzRef.readData(inputFile, testGroup+"/duzdz_ref");
19 // nastavenie potrebných parametrov simulácie
20 mParams.mNonUniformGridFlag = 0;
21 // vytvorenie kontajnera matíc
22 MatrixContainer mContainer;
23 mContainer[MI::kKappa].set(MT::kReal, reducedDims, kLoad, kNoCheckpoint, rootGroup+"/
    kappa_r_in");
24 mContainer[MI::kDdxKShiftNegR].set(MT::kComplex, DimensionSizes(reducedDims.nx, 1, 1),
    kLoad, kNoCheckpoint, rootGroup+"/ddx_k_shift_neg_r_in");
25 mContainer[MI::kDdyKShiftNeg].set(MT::kComplex, DimensionSizes(1, reducedDims.ny, 1),
    kLoad, kNoCheckpoint, rootGroup+"/ddy_k_shift_neg_in");
26 if (kIs3d)
27 {
28     mContainer[MI::kDdzKShiftNeg].set(MT::kComplex, DimensionSizes(1, 1, reducedDims.nz),
        kLoad, kNoCheckpoint, rootGroup+"/ddz_k_shift_neg_in");
29 }
30 mContainer[MI::kUxSgx].set(MT::kReal, fullDims, kLoad, kNoCheckpoint, rootGroup+"/
    ux_sgx_in");
31 mContainer[MI::kUySgy].set(MT::kReal, fullDims, kLoad, kNoCheckpoint, rootGroup+"/
    uy_sgy_in");
32 mContainer[MI::kUzSgz].set(MT::kReal, fullDims, kLoad, kNoCheckpoint, rootGroup+"/
    uz_sgz_in");
33 mContainer[MI::kDuxdx].set(MT::kReal, fullDims, kNoLoad, kNoCheckpoint, rootGroup+"/
    duxdx_in");
34 mContainer[MI::kDuydy].set(MT::kReal, fullDims, kNoLoad, kNoCheckpoint, rootGroup+"/
    duydy_in");
35 mContainer[MI::kDuzdz].set(MT::kReal, fullDims, kNoLoad, kNoCheckpoint, rootGroup+"/
    duzdz_in");
36 // pomocné a FFT matice
37 mContainer[MI::kTemp1Real3D].set(MT::kReal, fullDims, kNoLoad, kNoCheckpoint,
    kTemp1Real3DName);
38 mContainer[MI::kTemp2Real3D].set(MT::kReal, fullDims, kNoLoad, kNoCheckpoint,
    kTemp2Real3DName);
39 mContainer[MI::kTemp3Real3D].set(MT::kReal, fullDims, kNoLoad, kNoCheckpoint,
    kTemp3Real3DName);
40 mContainer[MI::kTempFftwX].set(MT::kFftw, reducedDims, kNoLoad, kNoCheckpoint,
    kCufftXTempName);
41 mContainer[MI::kTempFftwY].set(MT::kFftw, reducedDims, kNoLoad, kNoCheckpoint,
    kCufftYTempName);
42 mContainer[MI::kTempFftwZ].set(MT::kFftw, reducedDims, kNoLoad, kNoCheckpoint,
    kCufftZTempName);
43 mContainer[MI::kP].set(MT::kReal, fullDims, kNoLoad, kNoCheckpoint, kPName);
44 mContainer.createMatrices();
45 // vytvorenie inštalácie KSpaceFirstOrderSolver, nastavenie referencie na kontajner matíc
46 KSpaceFirstOrderSolver<kIs3d> solver;
47 solver.mMatrixContainer = mContainer;
48 // inicializácie FFT plánov; makrá START_OUTPUT_CAPTURE a STOP_OUTPUT_CAPTURE
49 // slúžia na potlačenie výpisov na stdout
50 START\_OUTPUT\_CAPTURE
51 solver.InitializeFftwPlans();
52 STOP\_OUTPUT\_CAPTURE;

```

```

53 // načítanie dát do kontajnera matic kontajnera
54 mContainer.loadDataFromInputFile();
55 // spustenie testovanej metódy
56 solver.computeVelocityGradient();
57 // porovnanie výsledkov s referenčnými hodnotami
58 EXPECT_TRUE(CompMat(duxdxRef, solver.getDuxdx(), 1e-5));
59 EXPECT_TRUE(CompMat(duydyRef, solver.getDuydy(), 1e-5));
60 if (kIs3d)
61 {
62     EXPECT_TRUE(CompMat(duzdzRef, solver.getDuzdz(), 1e-5));
63 }
64 }

```

Výpis C.3: Príklad jednotkového testu pre metódu computeVelocityGradient

Príloha D

Formát vstupného a výstupného HDF5 súboru

V nasledujúcich tabuľkách je uvedený formát vstupného a výstupného súboru. Formát vstupného a výstupného súboru je pre 2D a 3D simuláciu takmer rovnaký. Jediný rozdiel je, že vstupný/výstupný súbor pre 2D simuláciu neobsahuje skupiny dát pre dimenziu Z . Uvedené údaje sú prevzaté priamo z dokumentácie (Doxygen) projektu *k-Wave-Fluid-OMP*.

Tabuľka D.1: Formát vstupného súboru

Name	Size	Data type	Domain type	Condition
1. Simulation Flags				
ux_source_flag	(1, 1, 1)	long	real	Nz > 1
uy_source_flag	(1, 1, 1)	long	real	
uz_source_flag	(1, 1, 1)	long	real	
p_source_flag	(1, 1, 1)	long	real	
p0_source_flag	(1, 1, 1)	long	real	
transducer_source_flag	(1, 1, 1)	long	real	must be set to 0
nonuniform_grid_flag	(1, 1, 1)	long	real	
nonlinear_flag	(1, 1, 1)	long	real	
absorbing_flag	(1, 1, 1)	long	real	
2. Grid Properties				
Nx	(1, 1, 1)	long	real	
Ny	(1, 1, 1)	long	real	
Nz	(1, 1, 1)	long	real	
Nt	(1, 1, 1)	long	real	
dt	(1, 1, 1)	float	real	
dx	(1, 1, 1)	float	real	
dy	(1, 1, 1)	float	real	
dz	(1, 1, 1)	float	real	Nz > 1
3. Medium Properties				
3.1 Regular Medium Properties				
rho0	(Nx, Ny, Nz)	float	real	heterogenous
	(1, 1, 1)	float	real	homogenous
rho0_sgx	(Nx, Ny, Nz)	float	real	heterogenous

	(1, 1, 1)	float	real	homogenous
rho0_sgy	(Nx, Ny, Nz)	float	real	heterogenous
	(1, 1, 1)	float	real	homogenous
rho0_sgz	(Nx, Ny, Nz)	float	real	heterogenous and Nz > 1
	(1, 1, 1)	float	real	homogenous and Nz > 1
c0	(Nx, Ny, Nz)	float	real	heterogenous
	(1, 1, 1)	float	real	homogenous
c_ref	(1, 1, 1)	float	real	
3.2 Nonlinear Medium Properties (defined if (nonlinear_flag == 1))				
BonA	(Nx, Ny, Nz)	float	real	heterogenous
	(1, 1, 1)	float	real	homogenous
3.3 Absorbing Medium Properties (defined if (absorbing_flag == 1))				
alpha_coef	(Nx, Ny, Nz)	float	real	heterogenous
	(1, 1, 1)	float	real	homogenous
alpha_power	(1, 1, 1)	float	real	
4. Sensor Variables				
sensor_mask_type	(1, 1, 1)	long	real	file version 1.1 (0 = index, 1 = corners)
sensor_mask_index	(Nsens, 1, 1)	long	real	file version 1.0 always, file version 1.1 and sensor_mask_type == 0
sensor_mask_corners	(Ncubes, 6, 1)	long	real	file version 1.1 and sensor_mask_type == 1
5 Source Properties				
5.1 Velocity Source Terms (defined if (ux_source_flag == 1 uy_source_flag == 1 uz_source_flag == 1))				
u_source_mode	(1, 1, 1)	long	real	
u_source_many	(1, 1, 1)	long	real	
u_source_index	(Nsrc, 1, 1)	long	real	
ux_source_input	(1, Nt_src, 1)	float	real	u_source_many == 0
	(Nsrc, Nt_src, 1)	float	real	u_source_many == 1
uy_source_input	(1, Nt_src, 1)	float	real	u_source_many == 0
	(Nsrc, Nt_src, 1)	float	real	u_source_many == 1
uz_source_input	(1, Nt_src, 1)	float	real	u_source_many == 0 and Nz > 1
	(Nt_src, Nsrc, 1)	float	real	u_source_many == 1 and Nz > 1
5.2 Pressure Source Terms (defined if (p_source_flag == 1))				
p_source_mode	(1, 1, 1)	long	real	
p_source_many	(1, 1, 1)	long	real	
p_source_index	(Nsrc, 1, 1)	long	real	
p_source_input	(Nsrc, Nt_src, 1)	float	real	p_source_many == 1
	(1, Nt_src, 1)	float	real	p_source_many == 0
5.3 Transducer Source Terms (defined if (transducer_source_flag == 1))				
u_source_index	(Nsrc, 1, 1)	long	real	
transducer_source_input	(Nt_src, 1, 1)	float	real	
delay_mask	(Nsrc, 1, 1)	float	real	

5.4 IVP Source Terms (defined if (p0_source_flag == 1))				
p0_source_input	(Nx, Ny, Nz)	float	real	
6. K-space and Shift Variables				
ddx_k_shift_pos_r	(Nx/2 + 1, 1, 1)	float	complex	
ddx_k_shift_neg_r	(Nx/2 + 1, 1, 1)	float	complex	
ddy_k_shift_pos	(1, Ny, 1)	float	complex	
ddy_k_shift_neg	(1, Ny, 1)	float	complex	
ddz_k_shift_pos	(1, 1, Nz)	float	complex	Nz > 1
ddz_k_shift_neg	(1, 1, Nz)	float	complex	Nz > 1
x_shift_neg_r	(Nx/2 + 1, 1, 1)	float	complex	file version 1.1
y_shift_neg_r	(1, Ny/2 + 1, 1)	float	complex	file version 1.1
z_shift_neg_r	(1, 1, Nz/2)	float	complex	file version 1.1 and Nz > 1
7. PML Variables				
pml_x_size	(1, 1, 1)	long	real	
pml_y_size	(1, 1, 1)	long	real	
pml_z_size	(1, 1, 1)	long	real	Nz > 1
pml_x_alpha	(1, 1, 1)	float	real	
pml_y_alpha	(1, 1, 1)	float	real	
pml_z_alpha	(1, 1, 1)	float	real	Nz > 1
pml_x	(Nx, 1, 1)	float	real	
pml_x_sgx	(Nx, 1, 1)	float	real	
pml_y	(1, Ny, 1)	float	real	
pml_y_sgy	(1, Ny, 1)	float	real	
pml_z	(1, 1, Nz)	float	real	Nz > 1
pml_z_sgz	(1, 1, Nz)	float	real	Nz > 1

Tabuľka D.2: Formát výstupného súboru

Name	Size	Data type	Domain type	Condition
1. Simulation Flags				
ux_source_flag	(1, 1, 1)	long	real	
uy_source_flag	(1, 1, 1)	long	real	
uz_source_flag	(1, 1, 1)	long	real	Nz > 1
p_source_flag	(1, 1, 1)	long	real	
p0_source_flag	(1, 1, 1)	long	real	
transducer_source_flag	(1, 1, 1)	long	real	
nonuniform_grid_flag	(1, 1, 1)	long	real	
nonlinear_flag	(1, 1, 1)	long	real	
absorbing_flag	(1, 1, 1)	long	real	
u_source_mode	(1, 1, 1)	long	real	if u_source
u_source_many	(1, 1, 1)	long	real	if u_source
p_source_mode	(1, 1, 1)	long	real	if p_source
p_source_many	(1, 1, 1)	long	real	if p_source
2. Grid Properties				
Nx	(1, 1, 1)	long	real	
Ny	(1, 1, 1)	long	real	

Nz	(1, 1, 1)	long	real	
Nt	(1, 1, 1)	long	real	
dt	(1, 1, 1)	float	real	
dx	(1, 1, 1)	float	real	
dy	(1, 1, 1)	float	real	
dz	(1, 1, 1)	float	real	Nz > 1
3. PML Variables				
pml_x_size	(1, 1, 1)	long	real	
pml_y_size	(1, 1, 1)	long	real	
pml_z_size	(1, 1, 1)	long	real	Nz > 1
pml_x_alpha	(1, 1, 1)	float	real	
pml_y_alpha	(1, 1, 1)	float	real	
pml_z_alpha	(1, 1, 1)	float	real	Nz > 1
pml_x	(Nx, 1, 1)	float	real	
pml_x_sgx	(Nx, 1, 1)	float	real	
pml_y	(1, Ny, 1)	float	real	
pml_y_sgy	(1, Ny, 1)	float	real	
pml_z	(1, 1, Nz)	float	real	Nz > 1
pml_z_sgz	(1, 1, Nz)	float	real	Nz > 1
4. Sensor Variables (present if copy_sensor_mask)				
sensor_mask_type	(1, 1, 1)	long	real	file version 1.1 and copy_sensor_mask
sensor_mask_index	(Nsens, 1, 1)	long	real	file version 1.1 and sensor_mask_type == 0
sensor_mask_corners	(Ncubes, 6, 1)	long	real	file version 1.1 and sensor_mask_type == 1
5a. Simulation Results: if sensor_mask_type == 0 (index), or File version == 1.0				
p	(Nsens, Nt - s, 1)	float	real	-p or p_raw
p_rms	(Nsens, 1, 1)	float	real	p_rms
p_max	(Nsens, 1, 1)	float	real	p_max
p_min	(Nsens, 1, 1)	float	real	p_min
p_max_all	(Nx, Ny, Nz)	float	real	p_max_all
p_min_all	(Nx, Ny, Nz)	float	real	p_min_all
p_final	(Nx, Ny, Nz)	float	real	p_final
ux	(Nsens, Nt - s, 1)	float	real	-u or u_raw
uy	(Nsens, Nt - s, 1)	float	real	-u or u_raw
uz	(Nsens, Nt - s, 1)	float	real	-u or u_raw and Nz > 1
ux_non_staggered	(Nsens, Nt - s, 1)	float	real	u_non_staggered_raw (File version ==1.1)
uy_non_staggered	(Nsens, Nt - s, 1)	float	real	u_non_staggered_raw (File version ==1.1)
uz_non_staggered	(Nsens, Nt - s, 1)	float	real	u_non_staggered_raw (File version ==1.1) and Nz > 1
ux_rms	(Nsens, 1, 1)	float	real	u_rms
uy_rms	(Nsens, 1, 1)	float	real	u_rms

uz_rms	(Nsens, 1, 1)	float	real	u_rms and Nz > 1
ux_max	(Nsens, 1, 1)	float	real	u_max
uy_max	(Nsens, 1, 1)	float	real	u_max
uz_max	(Nsens, 1, 1)	float	real	u_max and Nz > 1
ux_min	(Nsens, 1, 1)	float	real	u_min
uy_min	(Nsens, 1, 1)	float	real	u_min
uz_min	(Nsens, 1, 1)	float	real	u_min and Nz > 1
ux_max_all	(Nx, Ny, Nz)	float	real	u_max_all
uy_max_all	(Nx, Ny, Nz)	float	real	u_max_all
uz_max_all	(Nx, Ny, Nz)	float	real	u_max_all and Nz > 1
ux_min_all	(Nx, Ny, Nz)	float	real	u_min_all
uy_min_all	(Nx, Ny, Nz)	float	real	u_min_all
uz_min_all	(Nx, Ny, Nz)	float	real	u_min_all and Nz > 1
ux_final	(Nx, Ny, Nz)	float	real	u_final
uy_final	(Nx, Ny, Nz)	float	real	u_final
uz_final	(Nx, Ny, Nz)	float	real	u_final and Nz > 1

5b. Simulation Results: if sensor_mask_type == 1 (corners)
and file version == 1.1

/p	group of datasets, one per cuboid			-p or p_raw
/p/1	(Cx, Cy, Cz, Nt-s)	float	real	1st sampled cuboid
/p/2	(Cx, Cy, Cz, Nt-s)	float	real	2nd sampled cuboid, etc.
/p_rms	group of datasets, one per cuboid			p_rms
/p_rms/1	(Cx, Cy, Cz, Nt-s)	float	real	1st sampled cuboid
/p_max	group of datasets, one per cuboid			p_max
/p_max/1	(Cx, Cy, Cz, Nt-s)	float	real	1st sampled cuboid
/p_min	group of datasets, one per cuboid			p_min
/p_min/1	(Cx, Cy, Cz, Nt-s)	float	real	1st sampled cuboid
p_max_all	(Nx, Ny, Nz)	float	real	p_max_all
p_min_all	(Nx, Ny, Nz)	float	real	p_min_all
p_final	(Nx, Ny, Nz)	float	real	p_final
/ux	group of datasets, one per cuboid			-u or u_raw
/ux/1	(Cx, Cy, Cz, Nt-s)	float	real	1st sampled cuboid
/uy	group of datasets, one per cuboid			-u or u_raw
/uy/1	(Cx, Cy, Cz, Nt-s)	float	real	1st sampled cuboid
/uz	group of datasets, one per cuboid			-u or u_raw
				and Nz > 1
/uz/1	(Cx, Cy, Cz, Nt-s)	float	real	1st sampled cuboid
				and Nz > 1
/ux_non_staggered	group of datasets, one per cuboid			u_non_staggered_raw
/ux_non_staggered/1	(Cx, Cy, Cz, Nt-s)	float	real	1st sampled cuboid
/uy_non_staggered	group of datasets, one per cuboid			u_non_staggered_raw
/uy_non_staggered/1	(Cx, Cy, Cz, Nt-s)	float	real	1st sampled cuboid
/uz_non_staggered	group of datasets, one per cuboid			u_non_staggered_raw
				and Nz > 1
/uz_non_staggered/1	(Cx, Cy, Cz, Nt-s)	float	real	1st sampled cuboid
				and Nz > 1
/ux_rms	group of datasets, one per cuboid			u_rms
/ux_rms/1	(Cx, Cy, Cz, Nt-s)	float	real	1st sampled cuboid

/uy_rms	group of datasets, one per cuboid	u_rms
/uy_rms/1	(Cx, Cy, Cz, Nt-s) float real	1st sampled cuboid
/uz_rms	group of datasets, one per cuboid	u_rms and Nz > 1
/uy_rms/1	(Cx, Cy, Cz, Nt-s) float real	1st sampled cuboid
/ux_max	group of datasets, one per cuboid	u_max
/ux_max/1	(Cx, Cy, Cz, Nt-s) float real	1st sampled cuboid
/uy_max	group of datasets, one per cuboid	u_max
/ux_max/1	(Cx, Cy, Cz, Nt-s) float real	1st sampled cuboid
/uz_max	group of datasets, one per cuboid	u_max and Nz > 1
/ux_max/1	(Cx, Cy, Cz, Nt-s) float real	1st sampled cuboid
/ux_min	group of datasets, one per cuboid	u_min
/ux_min/1	(Cx, Cy, Cz, Nt-s) float real	1st sampled cuboid
/uy_min	group of datasets, one per cuboid	u_min
/ux_min/1	(Cx, Cy, Cz, Nt-s) float real	1st sampled cuboid
/uz_min	group of datasets, one per cuboid	u_min and Nz > 1
/ux_min/1	(Cx, Cy, Cz, Nt-s) float real	1st sampled cuboid
ux_max_all	(Nx, Ny, Nz) float real	u_max_all
uy_max_all	(Nx, Ny, Nz) float real	u_max_all
uz_max_all	(Nx, Ny, Nz) float real	u_max_all and Nz > 1
ux_min_all	(Nx, Ny, Nz) float real	u_min_all
uy_min_all	(Nx, Ny, Nz) float real	u_min_all
uz_min_all	(Nx, Ny, Nz) float real	u_min_all and Nz > 1
ux_final	(Nx, Ny, Nz) float real	u_final
uy_final	(Nx, Ny, Nz) float real	u_final
uz_final	(Nx, Ny, Nz) float real	u_final and Nz > 1

Príloha E

Popis obsahu CD

Na CD nosiči sa nachádza nasledovná adresárová štruktúra:

1. **k-Wave-Fluid-OMP-2D/**

- adresár so zdrojovými súbormi prototypu 2D simulácie
- súbor `k-Wave-Fluid-OMP-2D/Readme.md` obsahuje kompletné inštrukcie ku kompilácii a spusteniu programu, popisuje obsah adresára
- adresár `k-Wave-Fluid-OMP-2D/Doxygen/` obsahuje kompletnú programovú dokumentáciu

2. **k-Wave-Fluid-OMP/**

- adresár so zdrojovými súbormi finálnej, zjednotenej 2D/3D simulácie
- súbor `k-Wave-Fluid-OMP/Readme.md` obsahuje kompletné inštrukcie ku kompilácii a spusteniu programu, popisuje obsah adresára
- adresár `k-Wave-Fluid-OMP-2D/Doxygen/` obsahuje kompletnú programovú dokumentáciu

3. **text/**

- adresár so zdrojovými súbormi textu práce

4. **Readme.md**

- textový súbor s popisom obsahu CD